Lestes C++ Compiler Documentation

Vojtěch Hála
egg

Jiří Kosina
jikos

Pavel Šanda
ps

Rudolf Thomas
Rudo

Miroslav Tichý
TMA

Petr Zika
pt

Jan Zouhar
jaz

May 2005

Contents

I Overview 15

1 Introduction 17
1.1 Project Assignment . . . 17
1.2 Project Status . . . 18
1.2.1 Using standard compiler constructing tools . . . 18
1.2.2 Systematic implementation of semantic analysis (by means of attribute grammars) . . . 18
1.2.3 Rigorous use of standard containers from STL and protective techniques of C++ . . . 18
1.2.4 Modern intercode representation (DAG) . . . 18
1.2.5 Strict separation and modularity of key blocks of the compiler . . . 19
1.2.6 Modular realization of all optional phases (optimizations) . . . 19
1.2.7 Independence of the data structures and algorithms on the target platform . . . 19
1.2.8 Providing a framework for building and extending the compiler . . . 19
1.3 Status of implemented and unimplemented features . . . 19
1.3.1 Preprocessor and lexer . . . 19
1.3.2 Syntactic analyzer . . . 20
1.3.3 Semantic Analysis . . . 20
1.3.3.1 Declarations . . . 20
1.3.3.2 Conversion sequences . . . 21
1.3.3.3 Expressions . . . 21
1.3.3.4 Statements . . . 21
1.3.4 Backend . . . 21
1.4 How to Read This Document . . . 22
1.4.1 Document Structure . . . 22
1.4.2 References into the C++ Standard . . . 22

2 Modules of Lestes Compiler 23
2.1 Overall General View of the Compiler . . . 23
2.2 Lexer and Preprocessor . . . 24
2.3 Syntactic Analysis . . . 24
2.4 Semantic Analysis . . . 24
2.5 Code Generation . . . 25

II Description of Data Structures 27

3 Preprocessor and parser structures 29
3.1 Character sets . . . 29
3.2 Characters . . . 29
3.2.1 Internal character type . . . 29
3.2.2 Manipulation . . . 29
3.2.3 Representation . . . 30
3.2.4 Strings . . . 30
3.3 Location . . . 30
3.3.1 File information . . . 30
3.3.2 Simple location . . . 30
3.3.3 Source location . . . 30
3.4 Tokens . . . 31

3.4.1 Character token . . . 31
3.4.2 Preprocessing token . . . 31
3.4.2.1 Token types . . . 31
3.4.2.2 Token values . . . 32
3.4.2.3 Token spelling . . . 32
3.4.2.4 Token flags . . . 32
3.4.2.5 Operations . . . 32
3.4.3 C++ token . . . 32
3.4.3.1 Token types . . . 32
3.4.3.2 Token values . . . 32
3.4.3.3 Literal information . . . 33
3.5 Token lists . . . 33
3.5.1 Token buffer . . . 33
3.5.2 Token sequence . . . 33
3.6 Macros . . . 34
3.6.1 Macro parameter list . . . 34
3.6.2 Macro expansion list . . . 34
3.6.3 Expansion list items . . . 34
3.6.4 Macro storage . . . 34
3.6.5 Macro arguments . . . 34
3.6.6 Taboo macros . . . 34
3.7 Syntactic token types . . . 35

4 Abstract Syntax Structures (AS) 37
4.1 Declarations . . . 37
4.1.1 Declaration Specifiers . . . 37
4.1.2 Declarators . . . 37
4.2 Names . . . 37
4.3 Statements . . . 38
4.4 Expressions . . . 38
4.5 Varia . . . 38
4.5.1 Wrappers . . . 38
4.5.2 Bearers . . . 38

5 Semantic Structures (SS) 39
5.1 Declarations . . . 39
5.1.1 Fields Common to All Declarations . . . 39
5.1.2 Alias Declarations . . . 39
5.2 Scopes . . . 39
5.2.1 Declaration Sequence Part . . . 40
5.2.2 Compound Statement Part . . . 40
5.2.3 Declaration Part . . . 40
5.3 Types . . . 40
5.3.1 ss_const, ss_volatile, ss_const_volatile . . . 40
5.3.2 ss_referential . . . 40
5.3.3 ss_reference, ss_pseudoreference . . . 40
5.3.4 ss_array . . . 41
5.3.5 ss_pointer . . . 41
5.3.6 ss_member_pointer . . . 41
5.3.7 ss_function . . . 41
5.3.8 ss_member_function . . . 41
5.3.9 ss_struct_base . . . 41
5.3.10 ss_class . . . 42
5.3.11 ss_union . . . 42
5.3.12 ss_typename_type . . . 42
5.4 Statements . . . 42
5.4.1 Compound Statement . . . 42
5.4.2 Declaration Statement . . . 43

5.4.3 Breakable Statements . . . 43
5.5 Expressions . . . 43
5.5.1 Unary Expressions . . . 44
5.5.2 Binary Expressions . . . 44
5.5.3 Function Call . . . 44
5.5.4 Assignment Expression . . . 44
5.5.5 Conversions . . . 45
5.5.5.1 Class ss_conversion . . . 45
5.5.5.2 Class ss_get . . . 45
5.5.5.3 Class ss_array_to_pointer . . . 45
5.5.6 Dereference . . . 45
5.5.6.1 Class ss_dereference . . . 45
5.5.7 Sideeffect . . . 45
5.5.8 Sequence Point . . . 46

6 Pseudo Instructions (PI) 47
6.1 Pseudoinstructions . . . 47
6.1.1 Generic pseudoinstruction . . . 47
6.1.2 Sequencepoint . . . 47
6.1.3 Leave function . . . 47
6.1.4 Load/Move/Store . . . 47
6.1.4.1 pi_abstract_move_st . . . 48
6.1.4.2 pi_abstract_move_dt . . . 48
6.1.4.3 pi_indirect_store . . . 48
6.1.5 Branches . . . 48
6.1.5.1 pi_unconditional_branch . . . 48
6.1.5.2 pi_conditional_branch . . . 48
6.1.5.3 pi_branch_multiple . . . 49
6.1.6 Binary pseudoinstructions . . . 49
6.1.6.1 Binary single type pseudoinstruction . . . 49
6.1.6.2 Binary double type pseudoinstruction . . . 49
6.1.7 Ternary pseudoinstruction . . . 49
6.1.7.1 Ternary single type pseudoinstruction . . . 49
6.1.7.2 Ternary double type pseudoinstructions . . . 49
6.1.8 Function calls . . . 50
6.2 Operands of pseudoinstruction . . . 50
6.2.1 Memory operands . . . 51
6.2.2 Non-memory operands . . . 51
6.2.2.1 Pseudoregister . . . 51
6.2.2.2 Literal . . . 51
6.3 Common structures . . . 51
6.3.1 pi_mem_factory . . . 51
6.3.2 pi_literal_info . . . 51

III Implementation 53

7 Preprocessor 55
7.1 Introduction . . . 55
7.2 Character level processing . . . 55
7.2.1 The character token filter . . . 55
7.2.2 Data sources . . . 55
7.2.3 Encoders . . . 56
7.2.3.1 Encoder implementations . . . 56
7.2.3.2 Encoder factory . . . 56
7.2.4 Character translation . . . 56
7.2.5 Line numbering . . . 56
7.2.6 Trigraphs . . . 56
7.2.7 Line joining . . . 56

7.3 Lexical analyser . . . 57
7.3.1 Flex scanner . . . 57
7.3.1.1 Processing extended characters . . . 57
7.3.2 Token buffering . . . 57
7.3.3 Include directive names . . . 57
7.4 Macro expansion . . . 57
7.4.1 Expander . . . 58
7.4.2 Operations . . . 58
7.4.2.1 Token concatenation . . . 58
7.4.2.2 Token stringification . . . 58
7.4.3 Object-like macros . . . 58
7.4.4 Function-like macros . . . 58
7.4.5 Expanding macros . . . 58
7.5 Preprocessing directives evaluation . . . 59
7.5.1 Include directive . . . 59
7.5.1.1 Unit part . . . 59
7.5.2 Conditional directives . . . 59
7.5.2.1 Condition stack . . . 59
7.5.3 Other directives . . . 59
7.5.3.1 Macro definitions . . . 59
7.5.3.2 Line control . . . 60
7.5.3.3 Miscellaneous . . . 60
7.6 Post-processing . . . 60
7.6.1 Preprocessing token filter . . . 60
7.6.2 Literal translation . . . 60
7.6.3 Space removal . . . 60
7.6.4 String concatenation . . . 60
7.6.5 Token conversion . . . 61
7.6.5.1 Literals . . . 61
7.7 Implementation specific issues . . . 61

8 Syntactic Analysis and Passing Data to Semantic Analysis 63
8.1 Introduction . . . 63
8.2 Overview . . . 64
8.3 Disambiguation Manager . . . 64
8.3.1 Disambiguation . . . 64
8.3.1.1 Details . . . 65
8.3.1.2 Nested disambiguation . . . 65
8.3.1.3 Delayed analysis . . . 65
8.3.1.4 Error recovery . . . 66
8.3.2 Internals . . . 66
8.3.2.1 Token reading . . . 66
8.3.2.2 Disambiguation control . . . 67
8.3.2.3 Spawning new analyzers . . . 67
8.4 Hinter . . . 67
8.4.1 Operation . . . 67
8.4.2 Interaction with other parts . . . 68
8.4.2.1 Qualification . . . 68
8.4.2.2 Elaborated specifiers . . . 68
8.4.2.3 Scope . . . 68
8.4.2.4 Set of found declarations . . . 68
8.4.3 Prefixer . . . 68
8.5 Parsers . . . 68
8.5.1 Disambiguation rules . . . 69
8.5.2 Look-ahead token issues . . . 70
8.5.2.1 Disambiguation at function scope . . . 70
8.5.3 Semantic values . . . 71
8.6 Token types . . . 71

8.7 Passing Data to SA . . . 71
8.8 To Be Implemented . . . 71

9 Lookup **73**
9.1 Lookup API . . . 73
9.1.1 Example . . . 73
9.1.2 Lookup Parameters in More Detail . . . 74
9.1.3 Filters . . . 74
9.1.4 Return Value . . . 74
9.1.5 Overview of Lookup Internals . . . 75
9.1.5.1 Filter Internals . . . 75

10 Structure Analysis **77**
10.1 SA Introduction . . . 77
10.1.1 Transformation and Analysis of AS Structures . . . 77
10.1.1.1 Parts of SA . . . 77
10.1.1.2 Common Terms of SA . . . 77
10.1.1.3 Level of a Sequence Point . . . 78
10.1.1.4 Conjugated Sequence Points . . . 78
10.2 SA Context . . . 79
10.2.1 AS Context Part . . . 79
10.2.2 SS Context Part . . . 79
10.2.3 SA Context Part . . . 79
10.3 Managing Contexts . . . 79
10.4 Statements . . . 79
10.4.1 Overview . . . 79
10.4.2 Parser actions . . . 80
10.4.3 Sequence points and other common contents . . . 80
10.4.4 Expression statement . . . 80
10.4.5 Return statement . . . 80
10.4.6 Compound statement . . . 81
10.4.7 Function body . . . 81
10.4.8 Declaration statement . . . 81
10.4.9 If statement . . . 82
10.4.10 While statement . . . 82
10.4.11 For statement . . . 82
10.4.12 Do statement . . . 83
10.4.13 Switch statement . . . 83
10.4.14 Continue statement . . . 83
10.4.15 Break statement . . . 83
10.4.16 Try and catch statement . . . 83
10.4.17 Labeled statement . . . 84
10.4.18 Goto statement . . . 84
10.4.19 Work to be done . . . 84
10.5 Declarations . . . 84
10.5.1 Class Declarations . . . 84
10.5.1.1 Class Visibility and Invisibility . . . 85
10.5.1.2 Accessibility of Class Members . . . 85
10.5.1.3 Base Classes . . . 86
10.5.1.4 Class Completion . . . 86
10.5.1.5 Declaration of Class Members . . . 86
10.5.1.6 Implementation . . . 87
10.5.2 Simple Declarations . . . 87
10.5.2.1 Overview of the Analysis Steps . . . 88
10.5.2.2 Handling of a Declaration Specifier Sequence . . . 88
10.5.2.3 Handling of Declarators . . . 88
10.5.2.4 Looking Up the Previous Declarations . . . 89
10.5.2.5 Introducing and Merging Declarations . . . 89

10.5.3 Usings and Aliases 89
10.5.3.1 Using Declarations 90
10.5.3.2 Implementation 90
10.5.3.3 Protective Aliases 90
10.6 Deconstruct SPSE 91
10.6.1 Overview 91
10.6.2 Implementation of expression transformation 91
10.6.3 The Decision Between Builtin Operator and Function Call 93
10.6.4 Return Value 94
10.7 Overload resolution 95
10.7.1 Overload resolution overview 95
10.7.2 File Structure 95
10.7.3 Programming Technique Used—Functionals 96
10.7.4 Implicit Conversion Sequences 96
10.7.5 Overload Resolution Itself 99
10.8 Work To Be Done 100

11 Transformation of SS to PI (ss2pi) 101
11.1 Introduction 101
11.2 Supporting structures & algorithms 101
11.2.1 Sequence points transformation 102
11.2.1.1 Implementation 102
11.2.2 Destructor tables 102
11.2.2.1 Structure of destructor tables 102
11.2.2.2 Implementation 103
11.2.3 Variable maps 103
11.2.3.1 Implementation 103
11.3 Statements transformation 103
11.3.1 Compound statement 103
11.3.2 Declaration statement 103
11.3.3 Expression statement 103
11.3.4 Return statement 104
11.3.5 If statement 104
11.3.6 While statement 105
11.3.7 Other loop statements 105
11.3.8 Goto statement 105
11.3.8.1 Destructor trees 106
11.4 Declarations transformation 106
11.4.1 Object declaration 106
11.4.1.1 Local-scope objects 106
11.4.1.2 Namespace-scope objects 107
11.4.1.3 Look-up 107
11.4.2 Function declaration 107
11.4.3 Structure declaration 107
11.5 Expressions transformation 107
11.5.1 Binary operators 108
11.5.2 Pointers, references and lvalues 108
11.5.3 lvalue to rvalue conversion 109
11.5.4 Dereference 109
11.5.5 var-ref 110
11.5.6 Assignment operator 110
11.5.7 Function calls 110
11.5.7.1 Parameters of funcall 110
11.5.7.2 Returning value 110
11.5.7.3 Type of funcall itself 110
11.5.7.4 Transformation details 111
11.5.8 Comparison operators 111
11.6 Literals 111

11.6.0.1 Literal_info transformation 111
11.7 Used backend interface in ss2pi 112
11.7.0.2 Passing pi structures to backend 112
11.7.0.3 Conversion of types to backend 112
11.7.0.4 Memory management 112
11.8 ss2pi transformation API 112
11.9 To be implemented summary 112
11.10 Transformations summary 113
11.10.1 ss layer to pi layer 113
11.10.2 Exception transformation 113

12 Code Generation **115**
12.1 Introduction 115
12.2 Directory Structure 115
12.3 Backend schema 115
12.4 Machine Description 116
12.4.1 XML Description File 117
12.4.1.1 DataType 117
12.4.1.2 Register 117
12.4.1.3 PseudoInstruction 118
12.4.1.4 Pseudoinstruction Operand 118
12.4.1.5 Groups, Templates 119
12.4.1.6 Assembly Language Elements 119
12.4.2 Generated C++ Code 119
12.4.3 Hand-Written Classes 120
12.4.3.1 Memory Allocators 120
12.4.3.2 Function Parameter Access 120
12.4.3.3 Symbol register 120
12.4.3.4 Data type ranges 120
12.4.3.5 Literal Information 120
12.4.3.6 Operand Output Code Generator 121
12.4.3.7 Pseudoinstruction Output Code Generator 121
12.5 Backend Processes 121
12.5.1 Passing input intercode structures to the backend 121
12.5.2 TU processing 121
12.5.2.1 Pseudoinstruction ordering 122
12.5.2.2 Target pseudoinstruction selection 123
12.5.2.3 Register allocation 123
12.5.2.4 Spill-code generation 123
12.5.2.5 Local Memory Layout Computation 124
12.5.2.6 Assembly code generation 124
12.6 Interfaces 124
12.6.1 Memory allocators 124
12.6.2 Function's parameter access 124
12.6.3 Literal loading 124
12.6.4 Copy-constructor call generation 124
12.6.5 Type information 125
12.6.6 Name mangling 125
12.7 Debugging 125
12.8 Things to Be Implemented 125

IV Framework **127**

13 Directory Structure **129**

14 Structure Generator **131**
14.1 Data Formats and Processing . . . 131
14.2 XML Validation . . . 131
14.3 Comments . . . 131
14.4 LSD File Header . . . 131
14.4.1 Root Element and XML Namespaces . . . 131
14.4.2 LSD Prologue . . . 132
14.5 Class Description . . . 133
14.6 Class Data Access . . . 134
14.7 Enumeration Description . . . 134
14.8 Type Description . . . 134
14.9 Doxygen . . . 134
14.10 Garbage Collection Support . . . 135
14.10.1 Marking Routine . . . 135
14.10.2 Smart Pointers . . . 135
14.10.3 Garbage-collectible Types . . . 135
14.11 Dumping . . . 136
14.12 Creating a Class Instance . . . 136
14.12.1 Constructor . . . 136
14.12.2 Factory Method . . . 136
14.13 Namespaces . . . 137
14.14 Visitors . . . 137
14.14.1 Basics . . . 137
14.14.2 Abstract and Concrete Visitors . . . 137
14.14.3 What Does the Generator Produce . . . 137
14.14.4 What Does the LSG User Have to Do . . . 138
14.14.5 LVD Files . . . 138
14.14.6 Implementing Visitors—The Inheritance Tree Cuts . . . 138
14.15 LSD Frequently Asked Questions . . . 139
14.15.1 Using-class or a Forward Declaration? . . . 139
14.15.2 Import or Implementation-import? . . . 139

15 Build System **141**
15.1 Basics . . . 141
15.2 Supported Targets . . . 141
15.3 Variables in Makefiles . . . 142
15.4 Variables Used for Linking . . . 143
15.5 Example Makefile . . . 143

16 STL Wrappers, Smart Pointers **145**
16.1 Supported Classes . . . 145
16.2 API . . . 145
16.3 Notes . . . 145
16.4 Pair . . . 146

17 Dumper **147**
17.1 Goals . . . 147
17.2 Achievement . . . 147
17.3 Assumptions . . . 147
17.4 API . . . 148
17.5 How a Dumper Examines Given Instance . . . 149
17.6 Examples . . . 149

18 Logger **151**
18.1 Usage . . . 151
18.1.1 An Alternative Approach to Logger Usage . . . 151
18.2 Configuration . . . 152
18.3 Implementation Notes . . . 152

19 Memory management **153**
19.1 Introduction . . . 153
19.1.1 Garbage collector . . . 153
19.1.2 Smart pointers . . . 153
19.1.2.1 Operations on pointers . . . 153
19.1.3 Collectible classes . . . 154
19.1.3.1 Construction . . . 154
19.1.3.2 Destruction . . . 154
19.1.3.3 Fields . . . 154
19.1.3.4 Methods . . . 154
19.1.4 STL wrappers . . . 155
19.1.4.1 Construction . . . 155
19.1.4.2 Destruction . . . 155
19.1.4.3 Comparators . . . 155
19.1.4.4 Operations . . . 155
19.1.4.5 Pair . . . 155
19.1.5 Templated classes . . . 156
19.2 Implementation . . . 156
19.2.1 Garbage collector structures . . . 156
19.2.1.1 Keystones . . . 156
19.2.1.2 Simple pointers . . . 157
19.2.1.3 Root pointers . . . 157
19.2.2 Garbage collector operation . . . 157
19.2.2.1 The marking method . . . 157
19.2.2.2 Mark phase . . . 157
19.2.2.3 Sweep phase . . . 157
19.2.3 STL wrappers . . . 157
19.2.4 Implementation issues . . . 158

20 Message reporting **159**
20.1 Introduction . . . 159
20.2 Reporting Interface . . . 159
20.2.1 Report structure . . . 159
20.2.2 Location . . . 159
20.2.3 Message . . . 159
20.2.3.1 Structure of message . . . 159
20.2.3.2 Message format string . . . 160
20.2.3.3 Message parameters . . . 160
20.2.3.4 Parameter formatters . . . 160
20.2.4 Defining messages . . . 160
20.2.4.1 LMD structure . . . 160
20.2.5 Reporting messages . . . 161
20.2.6 Building . . . 161
20.3 Reporting Implementation . . . 161
20.3.1 Introduction . . . 161
20.3.2 Structure . . . 161
20.3.3 Location . . . 162
20.3.4 Message oriented classes . . . 162
20.3.4.1 Message stencils . . . 162
20.3.4.2 Messages . . . 162
20.3.5 Reporting oriented classes . . . 162
20.3.5.1 Reporter . . . 162
20.3.5.2 Report filters . . . 162
20.3.6 Supporting entities . . . 163
20.3.6.1 Argument holders . . . 163
20.3.6.2 Overloaded operators . . . 163
20.3.7 LMD transformation . . . 163
20.3.7.1 Mapping of general tags . . . 163

20.3.7.2 Mapping of message tags . . . 163
20.3.8 Support in build system . . . 163
20.3.9 Further improvements . . . 164

V Executing Lestes 165

21 Compiling Lestes 167
21.1 System Requirements . . . 167
21.2 Compiling Lestes . . . 168

22 Executing Lestes binary 169
22.1 Compiling the assembler output. Linking the executable. . . . 170

23 Example source codes 171
23.1 LIO - Lestes IO library . . . 171
23.2 Example Sources . . . 171

VI Conclusion 173

24 Project History and Design Decisions 175
24.1 Analysis . . . 175
24.2 Development Framework . . . 175
24.2.1 Structure Generator . . . 175
24.2.2 Design patterns . . . 175
24.2.3 Memory management . . . 175
24.2.4 Build system . . . 176
24.3 Compiler . . . 176
24.3.1 Lexical analysis (preprocessor) . . . 176
24.3.1.1 External program . . . 176
24.3.1.2 Wrapped module . . . 176
24.3.1.3 Own implementation . . . 177
24.3.2 Syntactic analysis (parser) . . . 177
24.3.2.1 GLR . . . 177
24.3.2.2 Super-set of the language . . . 177
24.3.2.3 Hinter . . . 177
24.3.3 Semantic analysis . . . 177
24.3.4 Code generation . . . 178
24.3.4.1 Machine description . . . 178
24.3.4.2 Output Format . . . 178
24.3.4.3 ABI . . . 178

25 Conclusion and Future Work 179
25.1 Conclusion . . . 179
25.1.1 Doxygen documentation . . . 179
25.1.2 Internal notes . . . 179
25.2 Future Work . . . 179

VII Appendices 181

A Hierarchy of semantic structures (layer ss) 183

B Special Considerations in Semantic Analysis 189
B.1 Semantic Checks . . . 189
B.2 Rationales for Implementation Decisions . . . 194

C Lookup Algorithms **197**
C.1 Qualified Name Lookup . . . 197
C.1.1 Qualified Lookup Algorithm . . . 197
C.2 Koenig . . . 198
C.2.1 Finding of associated classes and namespaces for type T . . . 198
C.3 Lookup_class member access I . . . 200
C.4 Lookup - elaborated_type_specifiers, using_directive . . . 201
C.5 Lookup⇔ Hinter Interaction . . . 201

D Implementation definitions **203**
D.1 Reference(accessing expression through &) does not constitute an access to volatile variable. . . . 203
D.2 Dereference of pointer to volatile doesnot constitue an access to volatile object. . . . 203
D.3 Conversion of volatile lvalue(vol_get) to an rvalue constitutes an volatile access, hence a side effect. . . . 203

Part I

Overview

Chapter 1

Introduction

1.1 Project Assignment

The goal of the project is the creation of a C/C++ language compiler with an IA-32 (Intel 80386 compatible microprocessors) code generator. Such a goal is practically impossible to achieve by a student project. Therefore the main focus is to create a platform that can be used and extended by means of other software projects and/or master theses. This project does not aim to compete with commercial or Open-Source compilers, but to teach the participants of this and other projects to design and implement compiler components as well as to build a basis for experimenting with and evaluation of new optimization techniques. Despite this educative-experimental focus the outcome of this or one of the continuation projects should be an usable compiler.

The main guideline for the input language specification will be the ANSI Standard of the C++ language (ISO/IEC 14882); the extensions and/or limitations of other compilers will be disregarded. The goal of the project includes the construction of a C language compiler as well. The C compiler should use the C++ compiler internals to the maximum possible extent, and it should implement only the difference between the two languages. The main accent will be put on the implementation of key portions of the C++ language, the subsequent implementation of which could impact other parts of the compiler. Those portions are:

- namespaces
- multiple inheritance
- function overloading and user defined conversions
- virtual functions
- inline functions
- templates

The compiler architecture design focus will be to comply with demand for easy maintainability and extensibility. That comprises mainly of:

- using standard tools for lexical and syntactic analysis (flex, bison)
- systematic implementation of semantic analysis by means of attribute grammars
- rigorous standard STL containers and protective C++ techniques (e.g. intelligent pointers) use in the compiler data structures, irrespective of their efficiency
- use of modern intercode representation type (DAG) for information passing among the compilation phases to the maximum extent possible
- strict separation and easy replaceability of key compiler blocks
- modularization of all optional phases (optimizations)
- independence of data structures and algorithms on the target architecture

This project is targeted on IA-32 code. However, most parts of the compiler should be written in a target independent manner that includes changing the size of the basic types on 16 or 64 bit architectures. The compiler should be usable as a cross-compiler on any 32 bit or wider ANSI C++ platform.

The output of the compiler should comprise of a code in a standard calling and naming conventions. Concrete choice of target platform (win32/Unix), output format (object files/assembler source) and other details of the implementation will be adjusted to the preferences of the project participants, provided that the architecture shall provide an easy means of creation of other output modules. The project should implement one win32 and one GNU/Linux format and calling convention ideally.

The primary goal of the project is to achieve analytical part of the compiler completeness; the requirements on the quality of the generated code will be adjusted to the number and capacities of the participants ranging from wholly unoptimizing code generator to a quality generator containing a scheduler.

Taking in account the adherence to a standard calling convention it will be unnecessary to implement own library version; with a larger number of participants even reimplementation of base C libraries is possible. The reimplementation of higher libraries of the C++ language (STL and streams) can be included in one of the continuation projects.

1.2 Project Status

In this section a brief summarization of the work actually done will be presented. The project history and the important design decisions that were made during the project development, are described in detail in 24.

1.2.1 Using standard compiler constructing tools

This aim was achieved. The preprocessor contains C++ tokenizer and token concatenator, which are both implemented in `flex`. This choice turned out to support the idea of extensibility and maintainability. Other commonly used preprocessors use hand-written tokenizers which is error-prone and usually puts a lot of mundane work on the maintainer. Minor workarounds had to be deployed to fully support wide characters, as mandated by the Standard.

The parsers used in the syntactic analyzer are generated with non-modified version of `bison`. Despite the initial fears that the complex disambiguation mechanism will force tweaking the bison internals, it was not necessary after all. The only downside is that the generated parser is huge, and some compilers might refuse to process it.

1.2.2 Systematic implementation of semantic analysis (by means of attribute grammars)

This goal was achieved only partially. Many provisions of the standard averted us from more thorough application of attribute grammars to the semantic analysis. Some parts of the standard were formulated in an explicitly imperative manner, which was not deemed worthy to rework in a declarative way. Handling of which kinds of declarations, expressions and statements is discussed in more detail in 1.3. Source codes that demonstrates the correct behavior are provided together with the project source code.

1.2.3 Rigorous use of standard containers from STL and protective techniques of C++

While it was decided to use garbage collector to lift burden of memory management from the ordinary programmer it was impossible to use STL containers directly. Adapters were developed to bridge the incompatibilities between garbage collected objects and standard containers.

In the course of developing garbage collector, the techniques of "smart pointers" were extensively used, again not to burden the programmer with the use of garbage collection.

1.2.4 Modern intercode representation (DAG)

Several modern intercodes (Whirl (Open64), SSA, DAG) were explored and used as a model for our own intercode. At first it was decided to use a sort of SSA intermediate representation, which combines the power of DAG with ease of implementing future optimizations. This approach was abandoned due to SSA being too complex for our needs. Moreover our intercode representation shares one important aspect with SSA, namely each pseudoregister is written to exactly once which makes transition to the full-fledged SSA form easier, once the need to use such advanced intercode representation arises. The DAG-likeness was retained nevertheless.

1.2.5 Strict separation and modularity of key blocks of the compiler

The low stages of intercode are designed with source language independence in mind as well as the target machine independence. The upper layers of the intercode are not possibly language-neutral as they reflect the source code more closely.

The individual parts of compiler have well-defined interfaces, which makes it possible to replace individual parts with another implementation with corresponding interface, with reasonable effort, in the case such replacement is needed.

During the course of development, a helper component was made that allows to specify object classes in a language independent manner. However, only sample implementation generating C++ code was implemented.

1.2.6 Modular realization of all optional phases (optimizations)

While there were provided no optimization phases and the code generator is just a simplistic proof-of-concept one, it is conceivable that such modules would be inserted into the code generator, perhaps replacing part of it.

1.2.7 Independence of the data structures and algorithms on the target platform

The only machine-dependent structures are present in the backend part of the compiler. The backend provides interfaces, allowing other parts of the compiler to request target-platform-dependent data, if they need them (such as sizes of types for processing `sizeof()` operator).

1.2.8 Providing a framework for building and extending the compiler

In order to provide a rich and easy-to-use framework certain advanced techniques were employed. The repetitive and tedious task of memory management was alleviated by the use of smart pointers and a garbage collector. This, however, resulted in the need for special supporting routines in every class. In order to ease the use of these advanced techniques a tool was made that would automatically generate the supporting code for all classes. The XML language has been chosen as a suitable means to write a meta-description of the classes, their contents and their relations. The XML description was transformed to C++ code by special application written in the W3C consortium XSL transformations language. We have developed our own set of XML tags useful to describe the classes which is independent on the (objective) programming language used and supports some of the widely used design patterns such as visitors, factory methods, and singletons.

The extent of data structures describing the source program at the various levels of its transformation towards machine language precludes the effective use of standard debugging tools. The debugging support consists of two main parts.

The first is an advanced replacement for debug printouts, that allows us to observe the control and data flow of the compiler by allowing to enable, disable, and redirect the output of the loggers, that are organized in a tree-like fashion without the need of recompilation.

The second is the support for dumping arbitrary objects into a XML representation. The dumping support is capable to output two different kind of XML documents. One of the formats is suitable for tree-like structure viewing in XML editors. The second can be readily transformed into HTML by a tool that was developed. The HTML permits easy viewing of the dumps via a browser interface.

Due to the complexity of the data structures, the Doxygen-generated documentation is also an important source of information. Most hand written functions and methods contain doxygen compatible comments, as do all the generated classes and methods.

1.3 Status of implemented and unimplemented features

This section briefly summarizes what features are implemented by individual parts of the compiler. This should give basic overview what language constructs will pass which phase and also what is still to be implemented in individual compiler blocks.

1.3.1 Preprocessor and lexer

Supported features are

- source files in ASCII and UTF-8 encoding,

- extended characters,
- trigraph sequences,
- recognizing C++ tokens,
- including files,
- defining macros,
- expanding macros,
- conditional directives.

Not implemented are

- expression evaluation which has to be done in extended arithmetics,
- conditional `#if` and `#elif` directives, because they need the expression evaluator.

1.3.2 Syntactic analyzer

Parser itself supports all aspects of the language. However, as the semantic part does not provide corresponding features, the following constructs are not currently recognized by the syntactic analyzer:

- member functions
  - constructors
  - destructors
  - type conversion operators
  - operators
- templates
  - class templates
  - function templates

1.3.3 Semantic Analysis

This part provides basic overview of language constructs that are accepted by the semantic analysis.

1.3.3.1 Declarations

Supported features are

- object declarations,
- function declarations,
- typedef declarations,
- named namespace definitions,
- class declarations.

Not implemented are

- templates,
- arrays (due to necessary constant expression evaluation),
- `friend` declarations,
- constructor declarations,
- operator declarations,
- anonymous namespace definitions,
- default function arguments.

1.3.3.2 Conversion sequences

- arithmetic type conversions and integral promotions work (but not for const-volatile qualified variables. There are quite elaborate rules for such conversions prescribed by Standard and will take quite a long time to implement them, but there is no practical obstacle apart from lack of time).
  - Derived-to-base conversion of classes are implemented
  - Reference binding works. Binding of lvalue to reference works flawlessly throughout the whole compiler. Binding of rvalue to `const` reference generates proper intercode structures, but is not handled by backend yet. The same situation is currently with functions returning reference.
  - Pointers and basic pointer arithmetics works. Multi-level pointers don't work. This is related to non-working const-volatile qualification in conversions, as multilevel pointers and cv-qualification are tightly connected.
  - User-defined conversion functions and conversion-by-constructor are not implemented. The reason for this is the frontend is unable to handle constructors so far.

1.3.3.3 Expressions

- transformation of basic operators works for built-in types. This includes implementation of the following operators: `= , += -= *= /= %= ^= &= |= <<= >>= == != <= >= < > | & ^ << >> + - * / % || && ++ -- ~ ! (), sizeof()`
  - transformation of the following operators doesn't work yet (the reasons lie primarily in the fact that they corresponding structures are not handled by frontend yet): `this`, `typeid`, `delete`, `throw`, `new`, cast operators, member access.
- Overload resolution for overloaded functions works for parameter types in the terms of functionality of conversion sequences described above.

1.3.3.4 Statements

The following statements do compile correctly.

- Expression statement (`a=3; f(x); i++;` ...)
- Return statement (`return f(3); return;` ...)
- Compound statement (`{ }`)
- Declaration statement (`int i=3; class X;` ...)
- If selection statement (`if (int i=a+b) ...; else ...;`)
- While loop statement (`while (true) {...}`)

The `for` loop statement is correctly created and passed to the SS2PI part, but not handled there yet. Possible implementation of the other statements is suggested in 10.4. The statements not mentioned here are correctly parsed, but there are no semantic actions to handle them (details in 10.4). Therefore the non-implemented statements are ignored by the Lestes compiler without any error message.

1.3.4 Backend

Supported features:

- Machine description for i686 architecture only.
- Integral datatypes.
- Asm code can be generated for all the types of pseudoinstruction.
- Functions.
- Local variables including both static and dynamic initialization.

- Global variables and global variable initialisation with constant expression.
- Literals
- Branches.
- Name mangling and calling convention compatible with GNU gcc. This means that our code can be linked against objects generated with GCC compiler.

Unsupported features:

- Floating-point types
- Structural types (class/struct/union/array)
- Member and virtual functions
- Objects with static storage duration
- Functions with variable count of parameters
- Nested functions.
- Global variable initialisation with non-constant expression.

1.4 How to Read This Document

1.4.1 Document Structure

This document consists of six separate parts (each of them separated into several chapters).

The first part gives the basic overview about the implemented functionality of Lestes compiler (this chapter) and the overview description of individual basic blocks of the compiler.

The second part provides description of design of the data structures, which are used in Lestes compiler internals to represent the compiled source code in various stages of the transformation process (preprocessor data structures, Abstract Syntax (AS) data structures and Semantics Structures (SS)).

The third part provides detailed description of implementation of individual compiler parts, describing used algorithms, discussing in detail technical solutions of individual problems, contains description of API which each part of the compiler provides and contains rationales for the decision which have been made during developing the low-level features of the compiler. Sections in this part, devoted to the concrete problems, also provide a description of work that needs to be done to add eventual missing functionality.

The fourth part describes the framework that was developed to ease the development of such a complex project.

The fifth part describes command line interface to lestes and describes steps needed to compile C++ sources using Lestes. Also description of compilation of how to compile lestes

The last part provides conclusions and summary of the whole project.

In the Appendices there are caveats extracted from C++ standard, which are primarily intended to be used for the next development to improve semantic analysis. This implies that these notes are not easy to understand without deeper knowledge of the Standard and compiler internals.

Every section concerning implementation of compiler internals contains detailed description of missing features regarding that particular section.

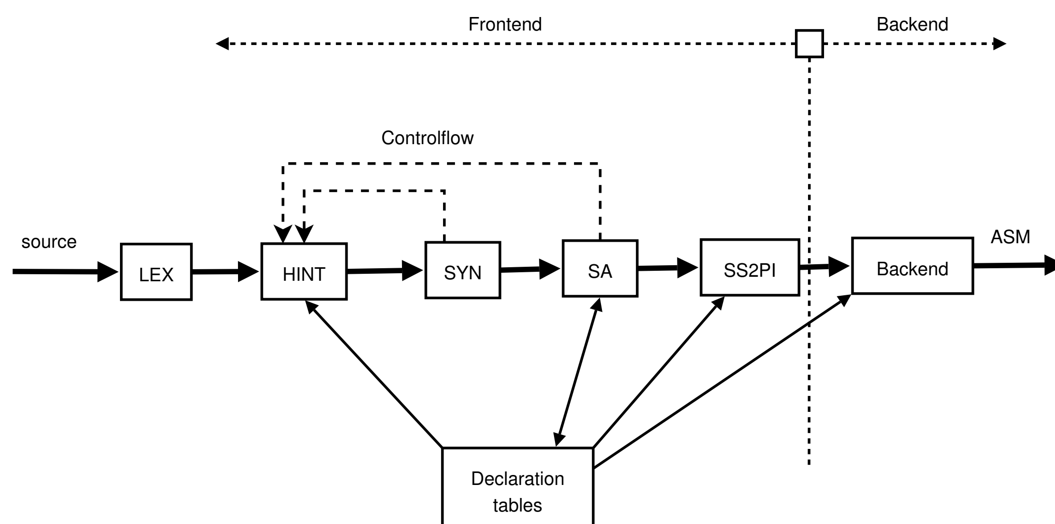
1.4.2 References into the C++ Standard

Throughout the whole documentation, specific chapters, sections and paragraphs in the C++ are quite often being referenced. These references have the form of the unique Chapter/Section/Paragraph pointer into the Standard, enclosed in brackets and printed in bold. For example **[8.5.2/3]** denotes paragraph 3 in section 8.5.2.

Chapter 2

Modules of Lestes Compiler

2.1 Overall General View of the Compiler



When a source program is being compiled by Lestes, it passes through several phases, or modules, of the compiler. The picture above gives a brief overview of interaction of these modules. Only the most important data and code flows are outlined here.

The LEX module comprises of a lexical analyzer and a preprocessor. It reads input files, processes preprocessor directives and expands macros. Its output is a stream of tokens.

Hinter (HINT) filters identifiers found in the token stream and tags them with type information that is looked up in the declaration tables. This information is of particular importance for syntactic analysis. The hinter is controlled from other modules as its mode of operation greatly depends on the context of currently processed language constructs.

SYN, the syntactic analyzer, analyzes language constructs and – given the hints – resolves ambiguities (constructs of the language that have more than one interpretation). It builds the AS structure and at certain points feeds it forwards to the semantic analysis.

The structure analysis (SA) processes data from syntactic analysis. It has two tasks: checking AS structures for errors that cannot be detected earlier; and filling the declaration tables and other SS structures such as statements and expressions. Together with SS2PI, it constitutes semantic analysis. SS2PI takes the SS representation and transforms them into pseudoinstructions, which are converted to assembler code in the backend.

The most notable fact is that a significant portion of frontend does not operate in phase-at-a-time manner, but rather in on-line fashion. This is mandated by the Standard, as it states that a name that is introduced into a scope is to be immediately seen by the syntactic analysis, for example. Therefore the hinter, parser, and semantic analysis are tightly interlinked and usually operate in lockstep on one declaration *before* tokens constituting the next one are read from preprocessor.

On the other hand, SS2PI and the backend operate on their own. They are given structures that represent the whole translation unit and transform them. When the transformation is completed, the next module is run to process the result as a whole.

2.2 Lexer and Preprocessor

When writing the preprocessor, the emphasis was put on implementing the features necessary for the rest of the compiler. They were namely recognizing properly all C++ tokens, handling extended character sets, integer and string literals and comments in the lexical analysis part. In the preprocessing part, defining and expanding macros was necessary, as was including other files and simple conditional inclusion (`#ifdef` directive family).

The supported input file encoding includes ANSI X3.4-1968 (7-bit ASCII) and UTF-8 encoded ISO/IEC 10646-1:2000 (also known as Unicode), as well as preliminary support for arbitrary 8-bit host-specific encoding which can be chosen at compile time. As dictated by the Standard, the preprocessor also recognizes extended characters represented as universal character names.

Certain aspects of the preprocessor were left out. Some minor issues, originating from unclear statements in the ISO Standard, were unconditionally solved according to one of the interpretations of the statements. The only module which was not implemented, is the preprocessor expressions evaluator, needed in `#if` and `#elif` preprocessing directives. The future implementation can be easily plugged into the existing code, provided that the necessary effort is invested into its development. The importance of the omitted features is not crucial to the compiler, as most inputs into the compiler are simple C++ programs, which do not make use of them.

The main disadvantage of the implementation is the lower performance of the resulting code. The reasons include avoiding hand optimization to keep the source code readable and maintainable, extensive use of debugging facilities and operating with unbuffered input. This disadvantage shall not be considered a drawback. The preprocessor operates in interactive mode and is not primarily intended for any performance-critical tasks.

2.3 Syntactic Analysis

Primary purpose of the syntactic analysis is parsing the language constructs. This involves disambiguation, a process of resolving the meaning of constructs that have more than one possible interpretation. Analyzed input is stored into abstract syntax (AS) structures. These are passed for further analysis to the semantic analyzers at specific times.

Unless ambiguities are handled completely in the semantic analysis (which is not the case in Lestes), a C++ syntactic analyzer cannot do without type information – in some contexts it needs to know whether an identifier denotes a type or a non-type. This information is provided by hinter. (Yet this is not enough to solve all ambiguities.)

Ambiguities of C++ are handled by pausing the main parser at places where ambiguous input is possible and analyzing incoming sequence of tokens by special parsers. When a resolution is found, the main parser continues to run knowing which of the options to pick. All parsers are bison-generated LALR(1) finite state automatons with stack.

As analysis of some parts of the source must be delayed, the analyzer is designed to be re-entrant. A new manager with its parsers and a hinter can be spawned at any time. Method bodies defined inside classes are good example of a constructs that cannot be analyzed immediately (see **[3.4.1/8]**).

The analyzer claims to cover the whole C++ language. All its ambiguities are handled as dictated by the Standard. However, this cannot be thoroughly tested, as the rest of the compiler does not implement the needed features (e.g. templates).

With regard to extensibility, there is not much room for the analyzer to be enhanced, given that C++ is not going to be changed dramatically, if at all. Error recovery was not concerned when designing the analyzer.

2.4 Semantic Analysis

The semantic analysis is constituted by two separate parts. The first part, called the Structure Analysis (SA), is responsible for transformation of AS structures into SS structures, which are, unlike the former, directly usable for determining the meaning of names. Type checking is performed and explicit representation for the implicit conversion sequences **[4], [13]** is introduced in the process of expression transformation. The choice of the best viable function is also made at that stage. The source program representation that springs as the result of the SA phase is devoid of any ambiguities, i.e. no aspect of the meaning is left to further determination. The expressions are reduced to a minimal set of operations, thus eliminating operator-assignment operator families, user defined operators (these are transformed to function calls), and increment/decrement operators. No information is lost during the process, except that the output of the prefix increment and decrement operators is barely distinguishable from their add-and-assign one and subtract-and-assign one parallels.[1] In a manner similar to expressions, the

[1] They can be told apart, because the literal one will have a different location information in both situations.

statements are reduced into a more rigorous, normalized form, that alleviates the burden of syntactic richness of the C++ language. E.g. the statement `while (int j = i--) foo(i, j);` is expressed as if it were written `{ int j = i--; while (j) { foo(i, j); } }`, except that the while destroys and initializes `j` before each iteration, not only before the first. This transformation makes possible to keep the following passes of the compiler as simple as possible, while simultaneously allowing them to be very flexible.[2]

The first part called the Structure Analysis (SA) is discussed in depth in chapter 10 on page 77.

The SA transformation of statements is performed partly by transforming the corresponding AS structures, partly by purely contextual means, and partly by the combination of both approaches. The AS structures are assembled in a straightforward manner during parsing, however it is sometimes necessary to perform the SA actions prior to the complete syntactic analysis of a C++ construct (e.g. a new scope for the compound statement needs to be created at the opening brace of the statement, while the AS representation of the compound statement is created only after the closing brace is encountered). Because of this, the clarity of the grammar has further diminished.

The SA is responsible for interfacing with the parser on its input side. The input consists of AS structures (see chapter 4). The outcome of SA consists of SS structures (see chapter 5).

The second part is called SS2PI. SS2PI converts *front-end* part to *back-end* part of the compiler. In terms of the Lestes project, this part is responsible for the conversion of the semantic structures (SS) to the backend structures (PI). That means, that after this conversion, there will be only pseudoinstructions with operands, which will be connected in a DAG-like structure, which is suitable for optimizations.

The separation between the SS and the PI intercodes is important in two ways: firstly, it allows the replacement of the front-end (e.g. the compiled language), while the backend part remains unchanged; secondly, it allows the replacement of the back-end part (e.g. target architecture or experimental optimizations on pseudocode), independently on the front-end part.

This separation is not total, namely as far as *declaration tables* access concerned. Our choice of a retargetable backend means that there is no way, how to determine the *target-type* of the declared *C++ type* without the backend. There are also dependencies, which cannot be settled by the backend itself. In both cases, special API is provided on both sides, so the proper actions remain on the appropriate side.

Moreover this transformation is not intended to make optimization over the structures, and a few cases, where the optimization could be done is documented in the detailed description of the SS2PI transformation (see chapter 11.)

2.5 Code Generation

The backend is as simple as possible. The basic goal is the ability to generate some code for simple C++ sources that is runnable on the target platform, without any architecture-specific optimizations, instruction scheduling, etc. Therefore, no optimizations are implemented and the generated code is by no means effective.

The backend takes intercode structures passed from the frontend (these structures represent a single translation unit). This bunch of interconnected structures is split into the blocks (function bodies) and transformed by backend to assembly language sources. At the end all assembler pieces are merged to a single output assembly file.

Transformations of the structures are performed by the backend modules. A module gets its input structures from the previous module, then it performs the transformation and finally passes the output to the next module. Implemented modules are the pseudoinstruction linearizator, the instruction selector, the register allocator, the spill-code generator, the stack layout computer and the asm code emitor.

Description of the target architecture registers, the supported data types, the instructions and the rules for instruction selection, the target asm language etc. is generated from the machine description XML file by XSLT generators.

The output asm file is then passed to an external asm compiler. It generates object file that can be linked with the libstdc++ supporting library by an external linker. In order to make the output asm linkable with external libraries, the backend tries to follow the ABI used by GNU GCC. For example, compatible symbol name mangling is implemented.

[2]Mainly, but not exclusively, by not precluding certain optimization possibilities.

Part II

Description of Data Structures

Chapter 3

Preprocessor and parser structures

3.1 Character sets

Chapter [2] states that a C++ compiler has to consider character sets. It has to provide mapping from the source file encoding into the basic source character set, using the universal-character-names for the nonrepresentable characters. The internal encoding must make no difference between the universal-character-name escape sequences coming from the mapping and written in the source. There are several character sets that had to be considered when implementing the Lestes compiler.

- The *build* encoding, in the environment that compiled Lestes. Ideally, this encoding should not interfere anyhow.
- The *host* encoding, in the environment that runs the compiler. All standard functions dealing with characters declared in header `<cctype>` operate with this encoding.
- The *internal* encoding, that is used to handle the character data inside the sources of Lestes.
- The *source* encoding, for the input into the compiler. It has to be converted into the internal encoding.
- The *target* encoding, in the environment that runs the programs compiled by Lestes. It has to be converted from the internal encoding for the output to operate correctly.

3.2 Characters

The internal representation and manipulation with characters is defined in `lestes/std/character.hh`. It introduces `hchar` representing the host character type, `ucn` (universal character notation) type and helper `character` class.

3.2.1 Internal character type

The `ucn` data type represents the internal character data type. It can contain a single character in either the internal or the external encoding. The internal encoding can represent all characters expressible through the universal-character-notation. The external encoding can represent characters in an unspecified encoding, which is context dependent. The data type is manipulated by the static methods defined in the class `character`.

3.2.2 Manipulation

The internal UCN can be created

- from the host encoded basic[1] character via `character::create_from_host()`
- from `ulint` value via `character::create_internal()`
- directly assigned a prepared constant `character::ascii_*`

[1] The basic characters as defined in the ISO Standard.

The external ucn can be created via character::create_external() only from ulint value.

The character class contains static methods, which substitute the standard character manipulation functions from <cctype>. They include predicates (is_basic(), is_digit()) and value transformers (to_host(), to_xdigit()).

3.2.3 Representation

The ucn is represented as a 32-bit unsigned integer. The highest bit is a flag distinguishing the character encoding. If the bit is cleared, the 31 bits contain an Unicode value, which is compatible with 7-bit ASCII. If the bit is set, the 31 other bits contain value of an externally encoded character. The 0xffffffff value is reserved as EOF value needed by the ucn_string template to represent end of file.

3.2.4 Strings

The strings are represented in classes inherited from STL basic_string template. String of hchar is represented by lstring type, string of ucn by ucn_string type. The ucn_string method to_host_string() converts the internal representation to host characters, using escape sequences \uXXXX and \uXXXXXXXX to express extended characters.

3.3 Location

Tracking the location of the created tokens is one of the tasks of the preprocessor. Each character in the translation unit has to obtain a location expressing its position in the inclusion chain and the translation unit part.[2] Because a single location is usually referenced from many objects and would be easily corrupted by a modification, all locations are read-only.

3.3.1 File information

The file_info class represents information about a part of a translation unit, which is either the toplevel source file, or any file included (transitively) into the source file. It contains

- the path and name of the file in host encoding,
- the origin of the #include directive that caused inclusion of the file.

3.3.2 Simple location

The simple location is used in the stages of processing, when the full source location information can not yet be known. The representation class simple_location contains only absolute physical line and column number. The information about the source file is implicit and need not be represented.

3.3.3 Source location

The source location represents complete information about location in the processed source files. It is passed from preprocessor to the rest of the compiler and used mainly for reporting messages to the user.

The class source_location class contains

- file information,
- line number within the file,
- column number within the line,
- order within the translation unit.

[2] I.e. the source file.

3.4 Tokens

Tokens are the key elements handled by the preprocessor. Apart from specific fields, tokens generally have three main properties.

- The *location* is the position of the token.
- The *value* is the (textual) information carried in the token.
- The *type* is the kind of the token, determining the interpretation of its value.

Like location, the tokens are read-only, because they can be referenced. There are three distinct token classes used in different stages of the preprocessor. Class `ucn_token`, for character tokens, `pp_token`, for preprocessing tokens and `cpp_token`, for the C++ tokens. All of them are inherited from the `basic_token` template.

3.4.1 Character token

The `ucn_token` is used in the character level processing. It contains only simple location, which has to be transformed to source location. The value is a single `ucn` character. The token types reflect the fact that in some phases the meaning of value is not yet known. The token type distinguishes

- characters before end of file (`TOK_NOT_EOF`),
- end of file (`TOK_EOF`),
- basic characters (`TOK_BASIC`),
- translated characters (`TOK_TRANSLATED`),
- error (`TOK_ERROR`).

The special type `TOK_ERROR` is used to pass a stored error message to the following stages of the preprocessing, where it can be reported with the the actual source location.

3.4.2 Preprocessing token

The preprocessing token is used in the macro and directive evaluation. It is created in the lexical phase. The location of `pp_token` is already `source_location`, but the order field is not set yet.

3.4.2.1 Token types

The token types reflect the token kinds as specified by the Standard ,

- header names (`<abc.h>, "abc.h"`),
- identifiers (`abc`),
- preprocessing numbers (`1ab2e-34`),
- string and character literals (`"abc", 'a'`),
- operators and punctuation (`#,+,*=,...`),
- other characters (`$`),

but moreover distinguishes several other kinds used internally,

- keywords (`int, while`),
- preprocessor keywords (`define`),
- boolean literals (`true, false`),
- blank,
- newline.

3.4.2.2 Token values

The value of preprocessing token is represented in `token_value` class, containing a `ucn_string`. Value is used for

- header names,
- literals,
- keywords,
- identifiers.

The `token_value` class is singletonized, each string is stored only once and referenced by all places of use. The comparison can thus be done through comparing pointers.

3.4.2.3 Token spelling

For the purposes of macro processing, each `pp_token` has its spelling, which is a textual representation of the token. The tokens with alternative spellings (the so-called *digraphs*) behave the same as ordinary tokens, except for their appearance, which is also considered when comparing macro definitions.

3.4.2.4 Token flags

Each token also has flags, to specify properties not dependent on token type. The flags are tested by predicates:

- `is_alternative()` set for operators with alternative spelling,
- `is_name()` set for identifiers and keywords from other tokens,
- `is_valued()` set for tokens that must have value specified.

3.4.2.5 Operations

Because the tokens are not modifiable, changes are done by cloning them with one of `clone*()` methods, which return new `pp_token` with some property changed. Another important operations are `equals()`, which tests equality and `congruent()`, which tests equality modulo location.

3.4.3 C++ token

The `cpp_token` represents the C++ programming language token. It is used to interface with the parser. It contains a fully specified `source_location`, including the order field. The `cpp_token` stream is created in the last stage of preprocessing, after all transformations of the source were carried out.

3.4.3.1 Token types

The token types are derived from the types of preprocessing tokens, after removing the types which have no meaning outside the preprocessor. The types comprise
identifiers,
keywords,
literals,
operators,
punctuators.

3.4.3.2 Token values

Like for preprocessing tokens, the `cpp_token` value is of the type `token_value`. It is set only for identifiers and literals. For the string and character literals, the content is translated to the target environment character set, for other literals and identifiers, it is kept in the internal encoding.

3.4.3.3 Literal information

Because all kinds of literals have the same token type, C++ tokens contain extra information carrying the properties of literals, extracted when checking the literal structure. The properties are represented by a class hierarchy based in `lex_literal`,

- `lex_integral_literal`,
- `lex_floating_literal`,
- `lex_character_literal`,
- `lex_string_literal`,
- `lex_boolean_literal`.

Each of the literals has own specific properties, for example

- integral number base (`123, 0123, 0x123`),
- number suffixes (`123UL, 3.14F`),
- string and character width (`'a', L'a'`),
- character length (`'abc'`)

which are later used to analyse the literal while transforming it into SS structures.

3.5 Token lists

The implementation needs to store the internal tokens in list-like structures. There are different demands on the lists for different token classes.

3.5.1 Token buffer

The `ucn_tokens` are stored into buffer of type `ucn_token_buffer`. The buffer works as a queue and is used to pass the tokens through the lexical phase. The method `add_back()` adds single tokens to the back of the buffer. At the other end, tokens can either be discarded completely by `advance()`, accessed by `peek_front()` to use their location for `pp_tokens`, or extracted to form `token_value` in different fashions (number, character and string literals) by the `extract_*()` methods.

3.5.2 Token sequence

The `pp_tokens` are stored into sequence of type `token_sequence`. The tokens can be accessed at both ends of the sequence via `peek_front()` and `peek_back()`. The reading methods represent the different regimes of usage

- `read()` reads the front token, no matter what,
- `read_front()` reads the token, but squeezes consecutive blanks into one,
- `read_front_skip_ws()` reads token after skipping all whitespace tokens.

The token sequence can also be cloned and compared for congruence of the content, which is necessary when checking macro redefinitions. Most importantly, the token sequence allows expanding macros. The method `expand_all()` expands all macros in the token sequence.

3.6 Macros

Macros are named entities, which can replace tokens coming from the lexical analysis. Preprocessor macros are represented by the `macro` class. There are *object-like* and *function-like* macros. The macro class contains

- name of the macro,
- location of definition,
- function-like flag,
- macro parameter list for function-like macros,
- macro expansion list.

3.6.1 Macro parameter list

The `macro_head` class represents the parameter list of a macro. It contains the parameters with the associated positions in the parameter list. All parameters must be name tokens with distinct spelling.

3.6.2 Macro expansion list

The `macro_body` class represents the expansion list of a macro, in the Standard referred to as replacement list. Its overloaded method `expand()` performs the expansion of the macro itself. The content of the class cannot be a simple token sequence to be replaced, because **[16.3]** specifies operations, which can be performed during macro expansion. Thus the expansion list consists of special items.

3.6.3 Expansion list items

The macro expansion list items are represented in class `macro_item`, which contains

- type of the action performed in the expansion,
- token sequence with tokens for the expansion,
- the index of a macro parameter, in case the action references some of the macro parameters.

3.6.4 Macro storage

The defined macros are stored in an associative container object of class `macro_storage`. Defining, undefining, redefining and searching for defined macros is covered.

3.6.5 Macro arguments

When parsing a function-like macro call, it is necessary to process the arguments passed to the macro. Each argument is represented as `macro_argument`, containing a `token_sequence` with the content of the argument. The argument can be empty. The class contains lazy evaluation methods used to obtain distinct flavors of the argument (plain, expanded, stringified). All arguments for a function-like macro call are stored in `macro_arguments` instance.

3.6.6 Taboo macros

To avoid infinite recursion when expanding macros, the macro `M` will not be considered when rescanning the tokens coming from the expansion of macro `M`, as stated in **[16.3.4]**. When the macro expansion contains nested macro calls, the list of macros not considered in expansion grows. To keep the track of the disallowed macros, each `pp_token` references an instance of `taboo_macros` class. It contains a set of macros, which are taboo when expanding the token. When a token is expanded as a macro, its instance of `taboo_macros` is inherited by the tokens in the expansions. The macro being expanded is always added to the new taboo set via a special `extend()` factory method, which caches the sets extended so far.

3.7 Syntactic token types

Syntactic analysis has its own set of tokens. Most of the time, they just wrap around `cpp_token`. They have additional properties needed for hints though. Packs, PAD and START tokens do not have their `cpp_token` counterparts. The most interesting is the identifier token.

Very brief overview:

- `bison_token` — base class for all tokens used in the analyzer; simple wrapper around `cpp_token`, its type for use with the parser is copied from the wrapped token; can contain user hint stored by the prefixer
  - `bison_id_token` — created in hinter; type is nominated by the hinter which also stores a set of found declarations inside
  - `bison_madeup_token` — used for tokens that have no correspondence in input, wrapped token is empty (null); used for PAD/START tokens
    - `bison_pack_token` — in addition to `bison_madeup_token`, contains a list of packed tokens (which all have dynamic type `bison_token`)

When `cpp_token` is read from the preprocessor, it is immediately wrapped in `bison_token` by the prefixer. If user hints are enabled, the prefixer handles them before the token enters the manager. The manager creates madeup and pack tokens, when requested by the parser. Instances of id token are created exclusively in the hinter, just before they are passed to the parser; they are never stored in the buffer in manager.

The reason behind not using `cpp_token` is that the syntactic analyzer has to create its own tokens with extended features, which would clutter the preprocessor code. In our implementation instances of `cpp_token` are created in the preprocessor, instances of `bison_token` and derived classes in bison. This also helps to clearly separate the modules.

For further details, see syntactic analysis documentation that begins at page 64.

Chapter 4

Abstract Syntax Structures (AS)

The AS acronym stands for Abstract Syntax. These structures represent the tree-like structure of the parsed program. The do not follow the actual grammar, however.

4.1 Declarations

The proper declaration representation pose a great challenge to the compiler, as they are both crucial to the meaning and difficult to handle. While the declarations need to be processed early, it is nearly impossible to get "hands on" a complete[1] declaration. The AS representation of the declarations is constructed nevertheless, because such construction proved as a great aid to developing and debugging the parser.

4.1.1 Declaration Specifiers

The declaration specifiers exhibit another instance of straightforward abstract grammar to AS structures mapping. Their handling during the SA phase is described in 10.5.2.2 in greater detail.

4.1.2 Declarators

The declarators are normalized in a way that eases their processing. Each declarator consists of an optional name accompanied by a possibly empty sequence of declarator operators. A declarator operator is a star (optionally accompanied by cv-qualifiers), an array operator (brackets with an optional size), an ampersand, or function parameter list.

Example: The declarator `*a(int)` is represented as `a,(int),*` whereas `b,*,(int)` corresponds to `(*b)(int)`. The details of declarator handling in SA is discussed in 10.5

4.2 Names

The representation of names in the AS came up as a result of long and painful evolution. The different needs imposed on the representation of names were working against each other. Thus precluding many of the more straightforward solutions. The current solution, i.e. the class `as_name` with it's supporting hoist of `as_id` subclasses provided us with all the necessary information we needed at later stages. The first remarkable feature of this approach is that the qualified and unqualified names are represented by the same structure. The second remarkable feature is that the names so represented can be of any kind, for example the same structure can accommodate a templatized operator name as well as an ordinary identifier.

The qualified names have the `was_qualified` field set to true whereas the unqualified have not. Irrespective of the status of this flag, the field `qualification` holds a reference to another as_name, which represent the scope, the identifier is either qualified into, or contained in.

The last remaining field of class `as_name` that was not discussed yet is the representation of the name itself expressed as an instance of `as_id` subclass. The diverse subclasses of as_id represent the full spectrum of C++ names allowed: plain identifiers, operator function ids, templated names, pseudodestructor names, constructor and destructor names. Each particular subclass contains all the necessary information to fully determine the name.

[1]Complete in the syntactic sense.

4.3 Statements

The statements are represented by a manner that directly follows their syntactic representation. The structure is discussed in the ISO standard in the Chapter **[6]** (q.v.).

The statements undergo little or no transformation during parsing and AS stage, almost no normalization takes place and the iteration statements for and while retain the representation of the condition as a special object different from the representation of expressions which takes them apart from the do iteration statement which does not permit the condition to be a declaration. The expression statement is not removed in cases that exhibit use of empty expression. The declarators in a simple declaration are not separated from each and from declaration specifier sequence, but are kept as a unit that is only subsequently separated into distinct objects representing the declaration. For detailed discussion of the representation of statements in the AS level structures please see the doxygen documentation.

4.4 Expressions

The expressions are represented by instances of as_expression subclasses. The precedence of operators expressed by the use of several nonterminal symbols in the concrete grammar is not directly observable, nor is the presence or absence of parentheses in the expression. The structure itself reflects the grouping of operands to operators. Thus the expressions a+b*c, a+(b*c), and ((a)+((((b))*(c))) are represented by the same structure.[2] The expression (a+b)*c is represented differently, so as to show the proper grouping expressed by the programmer.

The expression hierarchy on the AS level is formed along the arity of the operators represented. The binary operators do each have two operands while the unary have only one operand. Some of the expressions do not take other expressions as arguments. We can mention for example `sizeof` expression. Literals and identifier expressions alongside with `this` expression do not take any arguments because they represent sources of value, not the transformation of values. As with statements, more detailed information can be obtained from the doxygen documentation and the LSD files (see 14.1).

4.5 Varia

This section describes various structures that do not directly correspond to a source program construct but are nevertheless necessary for the implementation.

4.5.1 Wrappers

During the parsing it is necessary to pass sequences of AS structures back and forth. A set of wrappers for lists was developed that permits passing the lists as the semantic values of bison symbols without requiring error-prone modifications that would be otherwise necessary. The wrappers are manipulated by a set of macros that makes use of their common structure and generic algorithms of STL. That is used for underlying lists representation.

4.5.2 Bearers

For certain bison symbols, there is sometimes a possibility that they do not represent any part of the input text, while in other circumstances they do. A device for passing the semantic value in the second case needs to be adapted in order to convey the information about their incorrespondence with any portion of the source text in the source first. A trivial way how to achieve this would be to pass non-`NULL` and `NULL` pointer respectively, but the use of NULL pointers is precluded by the mechanism that is used for storing semantic value of the bison symbols. A new class is therefore introduced for each such situation, that contains the conveyed value or `NULL` but itself is not referenced by `NULL` pointer.

For example in the base-specifier of a class declaration, the keywords virtual, public, private and protected that are used to specify accessibility and kind of inheritance can be missing in the source program text. They are reduced by a bison rule that permits null derivation. The semantic value of the bison symbol representing that rule cannot be set to `NULL` however. So the classes `as_virtual_op_bearer` and `as_access_spec_opt_bearer` are introduced.

[2] Except, of course, for the location information.

Chapter 5

Semantic Structures (SS)

For quick overview of semantic structures hierarchy see appendix 5.

5.1 Declarations

The basic constituent of any C++ program is the declaration. A declaration introduces a new name (which is mostly an identifier) to a scope along with its meaning. All declarations are represented by instances of classes derived from class ss_declaration.

5.1.1 Fields Common to All Declarations

- `visible_since` - the timestamp of the first visible declaration
- `decl_time` - the timestamp of the declaration introduction
- `name` - the name being declared
- `contained_in` - the scope containing the declaration
- `type` - the type of the declaration
- `linkage` - mangling style and linkage (internal, external, none)

5.1.2 Alias Declarations

These declarations form a variety of subcategories. They have in common, that they contain a pointer to another declaration, which forms the "real" declaration.

- using declaration - the representation of the using declarations from source program
- fake declaration - a technical device for ensuring non-redeclarability
- injected class name declaration - a class name injected to the class

5.2 Scopes

The word "scope" is used in two, related senses. The first sense denotes a region of visibility for a name. The second denotes a region of a program, that contains declarations. These declarations become invisible when the region is left.

In the Lestes project the scopes (in the second sense) are represented by means of a declaration sequence (`ss_decl_seq`), compound statement (`ss_compound_stmt`), and declaration (`ss_declaration`) triple.

5.2.1 Declaration Sequence Part

The declaration sequence is an instance of the class `ss_decl_seq`. The instance holds a pointer to the parent declararion sequence, to the declaration that declared this scope, and to the compound statement part of the scope. Most importantly, in addition to the technicalities needed, it contains the declarations contained in that scope. The collection field holding the members of the scope is named `contents`. A list of ss_using_directive instances is available in the field named using directives. These are used in the lookup (q.v. in chapter 9).

5.2.2 Compound Statement Part

The compound statement part of the scope triple is used to hold declaration statements for the declarations declared in the scope. The declaration statements hold information necessary for initializing the declared entities. It is described in detail in .

5.2.3 Declaration Part

The declaration, pointed to by `ss_decl_seq::declared_by` field, is used to determine the kind of the scope.

5.3 Types

The types are represented by subclasses of class `ss_type`. All the subclasses are singletons thus providing straightforward means for type equality comparision. The structure more or less correspond to the English description of the types, e.g. "an array of ten integers" is reresented by an `ss_array` instance holding the number ten, and a pointer to the array base type represented by `ss_type_sint` instance.

The common abstract ancestor for all classes representing types in source code, is abstract class `ss_type`. This class holds fields and methods common for all successors of `ss_type`. These fields and methods are

- `is_void()` - this method returns bool variable, denoting whether the type is `void`. This is used in backend part of the compiler when generating pseudoinstructions, as the code generator has to distinguish between calling void function and calling function with return value.
- `is_volatile()` - this method returns bool variables, denoting whether the type is `volatile`. This information is needed whenever this variable is accessed (for example when performing lvalue-to-rvalue conversion, which is represented `ss_get`), as volatile variables require new sideeffect to be handled properly.

There is successor of `ss_type` called `ss_builtin_type`, which is again abstract class, which serves as ancestor for all types specified in C++ Standard as builtin (e.g. `char, short, int, long`, etc). There are other successors of the abstract `ss_type`, which are listed below

5.3.1 ss_const, ss_volatile, ss_const_volatile

All these types have, besides fields inherited from their abstract ancestors, the field named `what`, which is of the `ss_type` type, and indicates the type of the entity for which we hold const-volatile qualified type. These types (together with many other, as will be shown below), are implemented as singletons, which means that they contain `static` collection of elements, having the `ss_type` type. This collection is a map, so for each instance of the cv-qualification and the underlying type, there exists only one instance of such a structure.

5.3.2 ss_referential

This is abstract class serving as abstract ancestor for all types, having the referential nature.

This type can be also used for distinguishing between l-values and r-values, because the following holds:
`expression.type = ss_referential` ⇔ `expression is lvalue.`

5.3.3 ss_reference, ss_pseudoreference

These classes are successors of the abstract `ss_referential` class and they represent the reference (as defined in **[8.5.3]**) and pseudoreference (which is a term we used in lestes compiler for denoting all variables which are lvalues, but not references). They are formed in the similar shape as the const/volatile types mentioned

above, in the sense that they all contain the what field, which denotes which type the current variable is referencing/pseudoreferencing. They are also implemented as singletons, instances of which are differentiated by the ss_type of the underlying type.

5.3.4 ss_array

This type represents the array type in the source. It is also implemented as singleton, where the identical instances are distinguished by the map, indexed by the pair of the size and type. Arrays with unknown bounds are represented by using the value zero for their length, as it is otherwise impossible to have a zero-element array in the standard C++. Multidimensional arrays, or more correctly array of arrays are represented in a straightforward manner.

5.3.5 ss_pointer

This class represents a pointer to another type, contained in the what field. This type is also singletonized.

5.3.6 ss_member_pointer

This class represents the pointer to the class element, which is also called member pointer and it's semantics is defined in [8.3.3]. It holds the information about the base class and the field it is pointing to in fields base and item respectively. The member pointer has to be handled differently from the "usual" pointer, because there places in Standard, which define different behavior for pointers and member pointers. As an example, the conversions as per [4] are different for pointers and member pointers. Also when comparing the conversion sequences, as described in [13.3.3.2], member pointers and "usual" pointers have to be distinguished and handled differently.

5.3.7 ss_function

This type represents the function type. It is implemented as singleton as well, the instances are unified by return value and parameter types. The noteworthy fact about this type is, that there is defined ordering on these functions, for the purposes of declaration handling in frontend. The ordering considers (in this order) the first of the following to compare less: the ellipsis flag, the return value, and the parameter sequence. For the purpose of comparison, the value of the pointer determines the ordering for the objects representing data types, as these are all singletonized SS types and thus impose consistent ordering. The parameters are compared lexicographically in the usual manner, stopping at the first inequality. If there is none, the shorter sequence compares less. For the purposes of such comparisons, there are defined methods less_than(), which uses equal_signatures() method, comparing the equality of function signatures (i.e. prototypes). The ss_function class also carries the information, whether the function has ellipsis (i.e. incomplete parameter, specified by ... in arguments list. It was decided to hold this information as a field and do not create individual class for functions with ellipsis, as there is no C++ type, which would correspond to ellipsis.

5.3.8 ss_member_function

This type is descendant of the ss_function type and adds only one more field. This field is called this_type and is of the abstract type ss_type. It holds the type of the this field, as it will be seen inside the function. For example in the following code, the this_type will hold the structure representing "a const *" type.

```
class a {
          int f(int) const;
}
```

5.3.9 ss_struct_base

This type server as base abstract class for all structural types (which means struct, class, and union types). It contains pointer do ss_decl_seq, which holds declarations for all the members. The next notable field of this class is decl, which, when and instance of any successors (class, struct) is created, is pointed to the declaration of the respective class or struct. This is needed to allow access to enclosing scopes from inside the class and it's members. Information, whether the type is POD type (the short for "Plain-Old-Data"). This is mainly needed for backend, as the POD type shall occupy contiguous space. Last but not least, the structure holds the field called completion_time. This is very important in case of forward declarations. This field is set to

`ss_declaration_time::infinity()` unless the structure is completed, and then it is holding the completion time (of type `ss_declaration_time`).

5.3.10 ss_class

This class represents the class in compiled source code. It is direct descendant of the `ss_struct_base` class and contains `bases` and `descendants` fields, which are collections used to hold ancestors and descendants of the class in question respectively. Bases of the class are inserted into the list, because the ABI-structure-layout has to know the order. The descendants list contains list of direct descendants of the class. This field was added, because overload resolution process requires to know descendants of a class, to be able to perform conversion of member pointers to member pointers of derived classes, as specified in **[4.11/2]**.

5.3.11 ss_union

This class is successor of the `ss_struct_base` class and is used to represent union type. No extra fields are added, as the unions cannot have bases.

5.3.12 ss_typename_type

This is abstract class for template typename types and is prepared to represent the typenames in code such as

```
template< typename X > class A {
        typename X::T a(typename X::F);
};
```

However, since the templates are not yet implemented in Lestes compiler, processing this type will cause the compiler to emit error.

The field which is notable anyway, is the `qualified_name` as this field doesn't hold `ss_type` as one would expect, but it is of type `as_name` instead. This is because the fact that the actual type is not known before template initialization takes place. The conversion from `as_name` to `ss_type` can be done no sooner than the instantiation of the template is done.

5.4 Statements

Statements on this level are represented as descendants of abstract class `ss_statement`. It encapsulates fields common for all statements.

`location` Location of the statement in the source code.

`labels` List of labels pointing to this statement

`parent` Link to the scope containing this statement

`psp` Previous sequence point

`nsp` Next sequence point

`sequence_points` List of sequence points within this statement

Detailed information about how the SS structures for statements are filled in, can be found in 10.4. Detailed description of the structures is in Doxygen generated documentation. However, we describe some remarkable statements here.

5.4.1 Compound Statement

This statement, represented by class `ss_compound_stmt`, is a part of scope representation (5.2). It contains a reference to `ss_decl_seq` and a list of all SS statements in the scope. Additionally it contains an extra sequence point (`destructor_sp`) used for destruction of objects that were created within this compound. There is a flag `behavior` used by the SA to easily handle the statements containing inner compound statements such as `if`, `while`, and `for`.

5.4.2 Declaration Statement

Class ss_decl_stmt contains naturally a link to declaration. Additionally it contains a list of expressions (args) that were used used to initialize the declared object. Another field initializer_kind distinguishes the type of initialization. See **[8.5]** for details.

initializer_kind	Description	Example of such statement
IK_COPY_INITIALIZATION	Copy constructor.	int i=3; int a[3]={1,4,0};
IK_DIRECT_INITIALIZATION	Direct call to selected constructor.	int i(3); A i(3,3);
IK_DEFAULT_INITIALIZATION	Default constructor.	int i;
IK_NOT_APPLY_INITIALIZATION	Non-object declaration.	class A {}; int f();

The builtin types do not have true constructors, but the syntax is the same and so is the SS representation.

5.4.3 Breakable Statements

Breakable statements are: switch, for, while, do. The switch, for, and while statements contain a condition (as in if statement), which may be an expression or simple declaration according to **[6.4/1-3]**. The do statement cannot have a declaraton in condition. Each breakable SS statement also has a compound statement as its body. The body contains substatements. An example of the SS structure is shown in 10.4.11.

5.5 Expressions

Class ss_expression represents expression on the SS level. This class is a common ancestor for all the expressions. The expressions on the SS level unambiguously represent the meaning of the program, i.e. the operators resolve whether they are builtin or user-defined. Similarly the function called is selected from the set of overloadable functions candidates.

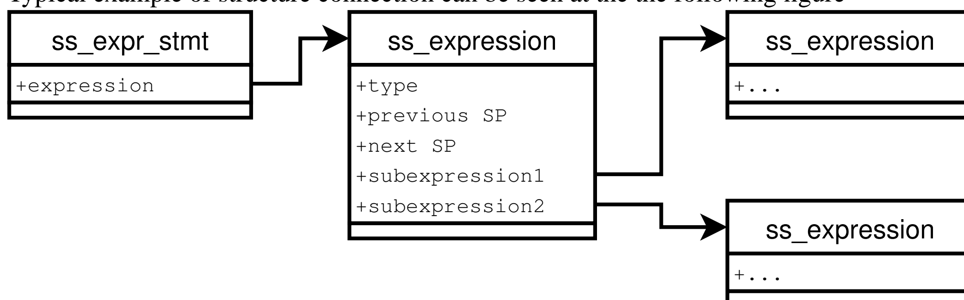
Generally, the expression representation consists of:

- *type,* represented by an instance of ss_type descendant (this is also used for determining lvalueness).
- *psp,* holding a reference to an sequence point that occurs before this expression is evaluated (mnemonic Previous Sequence Point)
- *nsp,* holding a reference to an sequence point that occurs after this expression is evaluated (mnemonic Next Sequence Point)
- references to sub-expressions (operands).

Expressions can basically be divided into two kinds - *lvalues* or *rvalues*.

This distinction is reflected in their type, by being a descendant of class ss_referential or not respectively. The only permissible lvalue expressions are instances of the ss_var_ref, ss_dereference, ss_assign, and ss_funcall descendants. The builtin operators that yield lvalues apart from the assignment operator (represented by ss_assign instance) do not have a separate corresponding SS representation. They are expressed by a structure consisting of other ss_expression instances mostly ss_assign accompanied with an arithmetic operation. This makes ++x, x+=1 and x=x+1 mostly equivalent in terms of SS representation.

Typical example of structure connection can be seen at the the following figure



Expressions may appear in a different contexts:

- expression statement

- initialization
- part of another statement (conditions, return, etc.)

5.5.1 Unary Expressions

These are the most simple regular expressions. They are directly derived from their source representation that exhibits the unarity as well. We rank `ss_gat, ss_neg, ss_bnot`, and others among them. The unary expressions expect and yield rvalues with the exception for certain special unary operations discussed elsewhere.

5.5.2 Binary Expressions

Binary expressions form the majority of arithmetic and logical expressions. As an example we can mention ss_add and ss_mul. Most binary expressions accept two rvalues of the same type and yield an rvalue of the same type. The exceptions to those are the relational expressions, that yield the type bool irrespective of the source operand types. The operator add that can sometimes accept as an operand an expression of pointer type and an expression of integral type as the other operand. Another notable exception is ss_sub, that yields integral result when supplied with two pointer operands.

The assignment operator ss_assign is discussed in detail in 5.5.4.

5.5.3 Function Call

There are several classes representing function call on the SS level. Those are differentiated in the manner the function is called. A different subclass is used for any of the following cases.

- call of non-static member function that is not virtual or a named virtual member function that is not called via VMT (explicitly qualified virtual function)
- call of virtual member function via VMT.
- call of a non-static member function via a member function pointer.
- call of a namespace level function or a static member function
- call of a namespace level function or a static member function via a function pointer.

Each of the function call types that call a non-static member function have an additional field representing the object argument passed to the member function. All the function calls have a list of arguments associated with them.

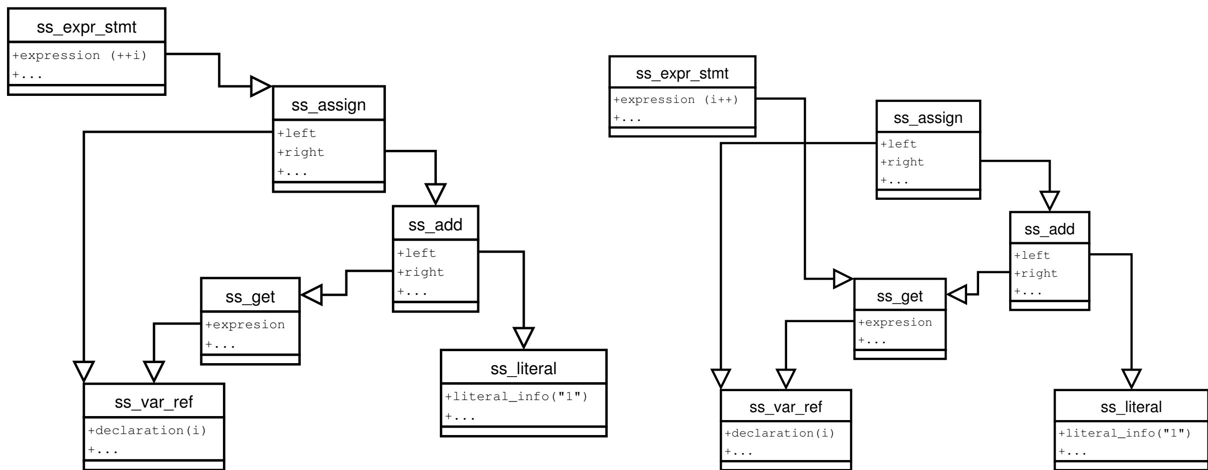
The function to be called is determined by a reference to a declaration in the case of direct function calls (i.e. function calls that are not performed via pointer or member pointer) by an expression in the remaining cases. Thus there is no unresolved ambiguity as to what function is to be called. All the ambiguities are already resolved by the overload resolution discussed in **10.7.**

5.5.4 Assignment Expression

The assignment expression represented by an instance of class `ss_assign` and expects an lvalue on it's left side and an rvalue on it's right side. It yields lvalue which is in a sense equivalent to the lvalue passed to it as a left operand. The difference between the lvalue of the left operand and the lvalue of the assignment expression is that the read of the lvalue passed to the assignment expression as the lvalue operand yields the value that was stored in the lvalue before the assignment expression has evaluated, while the lvalue obtained by reading the lvalue yielded by the assignment expression is the value just stored.

The "compound" operators (e.g. a += b, ..) are transformed on the SS level to a = (a+b).

The increment and decrement operators have to be treated specially, as can be seen on the following pictures (this is due to the differences between lvalues and rvalues of these expressions):



5.5.5 Conversions

Conversion as prescribed by the standard form a scattered category of expressions. Some conversions, mainly the arithmetic conversions and promotions, are expressed by instances of class ss_conversion. Some, such as lvalue-to-rvalue conversion, are expressed by different means (most notably ss_get). The user conversion are represented by ordinary function call expressions, and the reference binding is represented as an instance of yet another class family: ss_bind_reference and ss_bind_to_temporary depending on whether the value being bind is an lvalue or a temporary introduced for an rvalue respectively.

5.5.5.1 Class ss_conversion

This is the most basic conversion encountered in the C++ programs. It's structure resembles the structure of unary expressions, however the source type is also referenced in a designated field.

5.5.5.2 Class ss_get

The instances of class ss_get represent lvalue-to-rvalue conversion which is, when expressed in more familiar terms, a reading of a place in memory. The placement of ss_get instances is crucial to the following phases of translation. The type of the argument expression shall be of a referential type. The resulting type is stripped of any referential markers and in the case of non-class types also of the cv-qualification as the rvalues of non-class types shall not be cv-qualified.

5.5.5.3 Class ss_array_to_pointer

This class is a successor of the `ss_unary_expr` class, as it is a special case of `ss_conversion` discussed above. This class represents the conversion from array to pointer, and therefore has to be handled differently in the transformation to backend, but from the point of view of the SS structures, the `ss_conversion` and `ss_array_to_pointer` are the same.

5.5.6 Dereference

5.5.6.1 Class ss_dereference

The `ss_dereference` class represents unary `*` expression, having the semantics of the pointer dereference. As it's unary expression (having as the only argument the dereferenced pointer), it is descendant of the abstract `ss_unary_expr` class.

The builtin candidate functions for this expression are constructed in such a way, that the type of this expression is set to the type, which is obtained as a result of dereferencing the pointer (i.e. the type to what the pointer points).

5.5.7 Sideeffect

There are certain expressions, as prescribed by C++ standard, which have sideeffects. The most remarkable groups of expressions yielding sideeffects are all assignment expressions (including not only "plain" assignment

operation "=", but also "++", "–", "+=", etc.). The next notable expression yielding sideeffect is getting value (i.e. lvalue-to-rvalue) conversion from `volatile` type.

In some cases, the sideeffect of the expression is the only way how to "reach" the expression in the semantic tree, representing the source code. Let's assume the following example

```
a = (b = c, d);
```

In this example, the result of the comma expression, enclosed in brackets, is the expression `d` and therefore `d` is assigned into `a`. No other expression is therefore placed in the tree, representing the source code. If the expression standing on the left-hand side from the comma operator didn't yield sideeffect, it can be simply thrown away by the compiler, because it doesn't influence the behaviour of the resulting code by any means. However, in out situation, the expression is the assignment expression, which is expression with sideeffect (this sideeffect represents modifying the variable `b`). Therefore, this expression has to be evaluated.

In the previous paragraph, a basic overview of the sideeffect idea in C++ was demonstrated. The Semantic Structure, representing the sideeffect in the Lestes compiler, is called `ss_se`, and is direct descendant of the `ss_base_with_location` class. It hold references for previous and next sequence points (see later) and also reference to the expression from which this particular sideeffect originates, so that the expression could be evaluated.

The sequence points and sideeffects are mutually interconnected, signalizing the flow of time in which the expressions are to be evaluated, to allow the proper ordering of execution of expressions. The algorithm for achieving this is described in 11 and 12.5.2.1.

5.5.8 Sequence Point

The Standard provides a convenient device for describing semantics of C++ program, called sequence point. It is defined as a point in time when all sideeffects of the previous evaluations are complete and no sideeffects of the following evaluations have begun. The Standard also specifies places where these sequence points occur. The detailed description of the sequence point semantics and detailed discussion about sequence point semantics is written in 10.1.1.2. This section briefly describes the structure representing sequence point on the SS level.

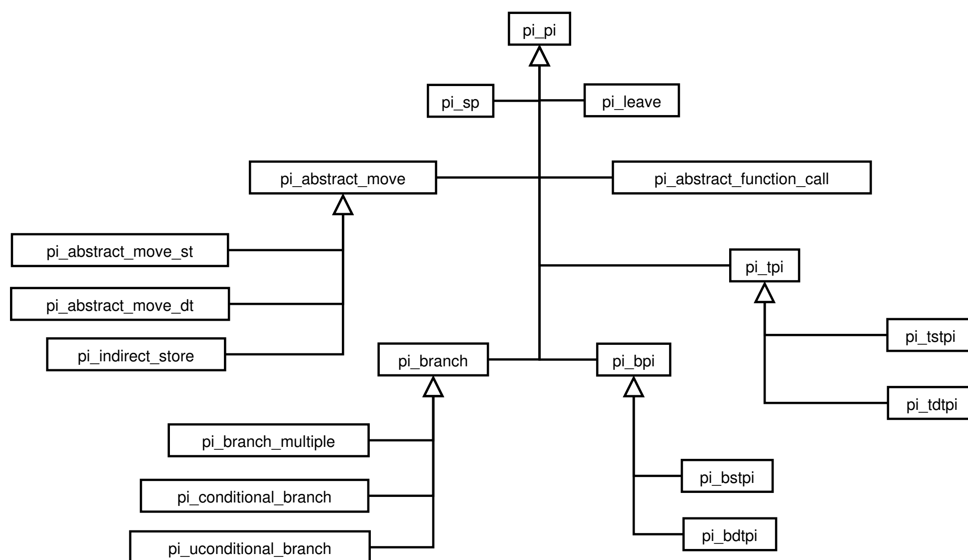
The class representing sequence point is called `ss_sp` and is a direct descendant of class `ss_base_with_location`. It contains in it's fields references to previous and next sequence points, and also integral number representing the sequence point level. The sequence points have to contain this information as backend uses it when performing topological ordering, to gain proper ordering of the instructions representing expression. This algorithm is discussed in detail in 12.5.2.1.

Sequence point structure also contains references for previous and next sideeffects, as the connection between sideeffects and sequence point is crucial for the sideeffects of the expressions to be evaluated properly.

Chapter 6

Pseudo Instructions (PI)

6.1 Pseudoinstructions



6.1.1 Generic pseudoinstruction

It is base for all pi-level pseudoinstructions. It has few fields that are common for descendant classes.

- `psp`
- `nsp`
- `level`

These information are used during linearization phase of backend.

6.1.2 Sequencepoint

`pi_sp` is pi-level representation of `ss_sp` 10.1.1.1.

6.1.3 Leave function

`pi_leave` represents return from function. It jumps to epilogue of the function where it is defined. Then it returns control-flow to caller.

6.1.4 Load/Move/Store

These pseudoinstructions are descendants of `pi_abstract_move`. They are divided into several subclasses:

6.1.4.1 pi_abstract_move_st

It represents load/move/st pseudoinstruction with one input and one output operands of the same type.

- `pi_mov` - moves data beetwen registers.
- `pi_ld` - loads data from memory to pseudoregister.
- `pi_ldv` - loads data from volatile memory to pseudoregister. It can't be optimized.
- `pi_ldi` - load literal to pseudoregister.
- `pi_st` - stores content of pseudoregister to memory.
- `pi_sti` - stores literal to memory.
- `pi_stiv` - stores literal to volatile memory. It can't be optimized.
- `pi_stv` - stores content of pseudoregister to volatile memory. It can't be optimized.

6.1.4.2 pi_abstract_move_dt

It represents load/move/st pseudoinstruction with one input and one output operands of different types.

- `pi_lda` - loads address of memory operand to pseudoregister.
- `pi_ldp` - indirect load. Loads content of memory at given address to pseudoregister.
- `pi_ldpv` - indirect load. Loads content of volatile memory at given address to pseudoregister. It can't be optimized.

6.1.4.3 pi_indirect_store

It is base class for indirect store via pointer.

- `pi_stp` - indirect store via pointer. It stores content of register to memory at given address.
- `pi_stpv` - indirect store via pointer. It stores content of register to volatile memory at given address. It can't be optimized.

6.1.5 Branches

These pseudoinstructions are descendants of `pi_branch`. They are divided into several subclasses:

6.1.5.1 pi_unconditional_branch

It represents unconditional jump. It jumps to labeled target without evaluation of any condition.

- `pi_ba` - jumps always.
- `pi_bn` - jumps never.

There are similar versions of the pseudoinstructions mentioned above that indirectly jumps to address.

6.1.5.2 pi_conditional_branch

Represents conditional jump. It jumps to labeled target if condition is satisfied. Condition is result of a previously called `pi_cmp` pseudoinstruction stored in register. For example:

- `pi_be` - jumps if the result of cmp says that compared operands are equal.
- `pi_bnl` - jumps if the result of cmp says that first of compared operands is not less than second one.

There are similar versions of the pseudoinstructions mentioned above that indirectly jumps to address.

Condition can be also given as bool value.

- `pi_bt` - jumps if value is true.
- `pi_bf` - jumps if value is false.

6.1.5.3 pi_branch_multiple

Conditional branch that jumps to one of multiple targets. Chosen target is the one that is associated with integral literal that is equal to switch value.

- `pi_bm` - targets are sequencepoints.
- `pi_ibm` - targets are addresses.

6.1.6 Binary pseudoinstructions

Descendants of `pi_bpi` have two operands.

6.1.6.1 Binary single type pseudoinstruction

`pi_bstpi` has two operands of the same type.

- `pi_neg` - destination operand gets negation of source operand.
- `pi_gat` - destination operand gets gation(identity) of source operand.
- `pi_bnot` - destination operand gets bitwise NOT of source operand.
- `pi_lnot` - destination operand gets logical NOT of source operand.

6.1.6.2 Binary double type pseudoinstruction

`pi_bdtpi` has two parameters of different types.

- `pi_cvt` - converts source operand to the destination type and stores result in destination operand.

6.1.7 Ternary pseudoinstruction

Descendants of `pi_tpi` have one output and two input operands.

6.1.7.1 Ternary single type pseudoinstruction

All operands of `pi_tstpi` are of the same type. Descendants of this class are arithmetical pseudoinstructions.
For example:

- `pi_add` - adds second operand to first one and stores result to output operand.
- `pi_land` - preforms logical AND of first operand and second one and stores result to output operand.
- `pi_bor` - preforms bitwise OR of first operand and second one and stores result to output operand.

6.1.7.2 Ternary double type pseudoinstructions

Output operand of `pi_tdtpi` is not of the same type as input operands.

- `pi_cmp` - compares input operands and result stores to output operand. The output operand is of special type that is designated to hold conditional codes.

Other descendants of this class are conditional set pseudoinstructions. They set output operand of type bool to true condition is met. False otherwise.
For example:

- `pi_sbe` - sets true if input operands equal.
- `pi_sbnl` - sets true if first input operand is not less then second one.

6.1.8 Function calls

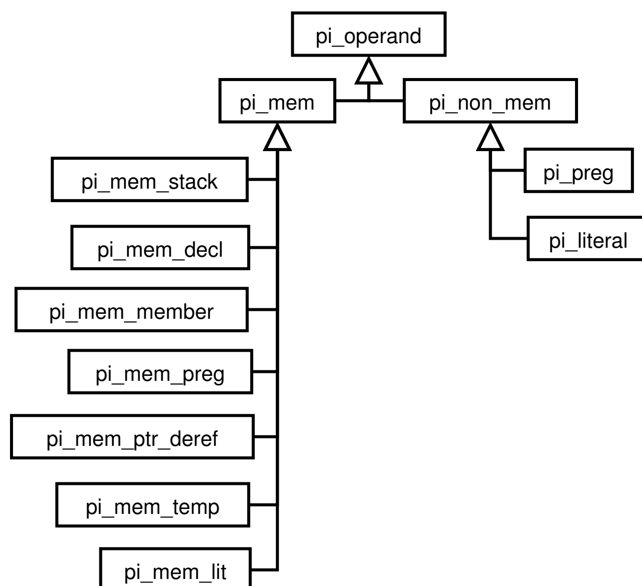
pi_abstract_function_call has few fields that are common to all function calls:

- args - list of function's parameters. Secret parameters for "this" value and address of space for return value passing are not added in this list.
- this_arg - secret "this" parameter. NULL if the called function is not member function.

Function calls:

- pi_call - call to function that returns non-void datatype. It has these fields:
  - type - type of return value.
  - rv - return value operand.
  - f - ss_declaration of the function 6.
- pi_callv - call to function that returns void datatype. It has these fields:
  - f - ss_declaration of the function 6.
- pi_icall - indirect call to function that returns non-void datatype. It has these fields:
  - type - type of return value.
  - rv - return value operand.
  - f - pseudoregister that holds address of the function.
- pi_icallv - indirect call to function that returns void datatype. It has these fields:
  - f - pseudoregister that holds address of the function.

6.2 Operands of pseudoinstruction



pi_operand has the following fields:

- origin - pseudoinstruction that assigns value to the operand. This field is NULL for pi_literal.
- type - data type of operand.

6.2.1 Memory operands

pi_mem represents reference to memory space (memory space is represented by `pi_mem_factory` class 6.3.1).
Memory operands:

- `pi_mem_stack` - references memory at function's stack.
- `pi_mem_decl` - references memory in global data section.
- `pi_mem_temp` - references temporal memory space.
- `pi_mem_member` - references data member of a structure.
- `pi_mem_lit` - memory that holds managed literal.
- `pi_mem_preg` - references pseudomemory in register.
- `pi_mem_ptr_deref` - references memory that is placed at specific address stored in pseudoregister.

6.2.2 Non-memory operands

6.2.2.1 Pseudoregister

`pi_preg` is virtual register. Count of pseudoregisters is not limited.

6.2.2.2 Literal

`pi_lit` represents literal. Its value is described by `pi_literal_info` class 6.3.2.

6.3 Common structures

6.3.1 pi_mem_factory

It represents space in memory.
Methods:

- `ptr<pi_mem> get_mem(ptr<pi_pi> origin)` - returns `pi_mem` object that references the memory and sets its origin.

There are `pi_mem_factory` subclasses. Each subclass corresponds to pi_mem class that references it:

- `pi_mf_stack`
- `pi_mf_decl`
- `pi_mf_member`
- `pi_mf_preg`
- `pi_mf_ptr_deref`
- `pi_mf_lit`

6.3.2 pi_literal_info

It holds informations about literal value and data type. It has two subclasses:

- `pi_li_simple_base` - it is literal of builtin data type (e.g. unsigned integer, signed char).
- `pi_li_compound_base` -it is composed of several simple literals (e.g. string “Hello” is composed of char literals 'H','e','l','l','o').

Part III

Implementation

Chapter 7

Preprocessor

7.1 Introduction

The C++ preprocessor is a part of the front-end of a C++ compiler. It is described mainly in the chapters **[2]** and **[16]** of the ISO Standard. The processing has sever stages, character level processing, lexical analysis, macro expansion, directives evaluation and post processing.

7.2 Character level processing

Before the input is passed to lexical analyser, the preprocessor has to read in and transform the input characters, as described in **[2.1/1,2]**. The whole process is managed by the `pre_lex` class. The operations comprise, in this order,

reading characters from the input (`data_source`)
source character mapping (`encoder`),
extended character translation (`special_tokens`),
line numbering (`line_numbers`),
trigraph translation (`trigraphs`),
line joining (`line_join`).

Unlike the statement of the Standard, the extended character translation is done before trigraph translation. As the Standard allows the phases to interleave, when the behavior is attained, this implementation is correct, because the translation of extended characters does not interfere with trigraph translation. All classes performing the operations inherit from `ucn_filter` class.

7.2.1 The character token filter

The `ucn_filter` class is an interface, representing abstract filter of `ucn_token`s. The constructor `ucn_filter::ucn_filter(ptr<ucn_filter>)` allows a filter to have another filter as its input. The interface contains `read()` method, returning the next `ucn_token` from the stream. The data from the input filter is accessed through protected `read_input()` method. The token filters can be chained via inputs to operate as a monolithic block.

7.2.2 Data sources

The `data_source` class is an interface, representing abstract source of binary data. This incoming binary data is already packed in the form of `ucn_token`s. The `ucn` value is externally encoded character, whose encoding is dependent on the data source encoding. As the value of the tokens was not interpreted yet, the token type is set `TOK_NOT_EOF`. The `TOK_EOF` is passed at end of input. The classes implementing the interface are

- `string_source`, taking data from host characters stored in `lstring`,
- `stream_source`, taking data from `::std::istream` input.

7.2.3 Encoders

To be able to interpret the data stream as characters, it is necessary to encode the source encoded characters as internal characters. The `encoder` class is an interface, representing abstract encoder into the internal character set. The type of coming from encoder is `TOK_NOT_EOF`, but the `ucn` value is already internally encoded.

7.2.3.1 Encoder implementations

The classes implementing the interface are distinguished by the source encoding,

- `encoder_host`, recognizing basic host characters,
- `encoder_ascii7`, recognizing 7-bit ASCII characters,
- `encoder_utf8`, recognizing UTF-8 encoded Unicode characters.

If any of the encoders encounters a value on its input, which is not valid for the respective encoding, the character is not encoded and `TOK_ERROR` with the appropriate message is returned.

7.2.3.2 Encoder factory[1]

The encoders are managed by `encoder_factory` class. This singleton class contains mappings from names to the available encoders. It allows creating individual encoders by specifying their associated name. The interface contains methods

- `insert()`, inserting mapping of a name to an encoder,
- `remove()`, removing an existing mapping,
- `summon()`, returning a new instance of an encoder by name.

7.2.4 Character translation

The class `special_tokens` performs checking of token values to assure, that they are not out of range specified for the universal-character-names. It also assigns more specific token types to the correct tokens. There is a counter watching for `TOK_ERROR` tokens coming from the `encoder` to avoid the flood of errors caused by wrongly chosen encoding.

7.2.5 Line numbering

The `line_numbers` class assigns each token a physical line number (starting from one) and a column number, making an entry of `simple_location`. These numbers should ideally reflect the position of the characters in the source file, as it would be recognized by a text editor, i.e. the numbers show positions of characters and not bytes forming the characters. Assigning the location has to be done before proceeding to the next stage, for already the trigraphs can collapse into new characters, whose column number would by no means be sequential.

7.2.6 Trigraphs

The trigraph sequences, starting with prefix `"??"` **[2.3/1]** are translated by the `trigraph` class. This has to be done before line joining, because the sequence `"??/"` translates to ”\” character used to designate joined lines. For any triple not mentioned as trigraph sequence, the output is left as an unchanged copy of input.

7.2.7 Line joining

Each sequence of backslash and newline is removed from the stream by the `line_join` class. When end of stream is encountered after backslash, both tokens remain unchanged, possibly rendering the program ill-formed in later stages due to the stray backslash. The sequence of backslash and newline immediately followed by end of file reports an error, while keeping the newline in the stream to suppress further error messages.

[1] The encoder factory is not currently active and `encoder_ascii7` is used defaultly.

7.3 Lexical analyser

The input of the lexical analyser is a stream of ucn_tokens, which is transformed to stream of pp_tokens. This process is encapsulated in the pp_lex class. This class also manages setting the proper location for the created tokens. The simple_location is transformed to source_location in the line_control class, considering the changes caused by the effect of the #line directive.

7.3.1 Flex scanner

As dictated by the Project Assignment, a scanner generator tool (flex) was used to write the lexical analyser. The lexical analysis is performed by pp_lex_guts.l scanner. The generated C source code pp_lex_guts.yy.cc is included into a separate package pp_lex_guts in pp_lex.cc, so that the identifiers do not poison the global namespace and do not collide with the other scanner (concat_guts.l). The pp_lex class manages switching internal scanner buffers and dispatches the calls to the scanner main processing method lexer_parse(), which returns individual pp_token instances.

7.3.1.1 Processing extended characters

The flex scanner generator, used to operates on host character set and does not support any extended characters. It was necessary to recognize the ucn, representing the basic source characters and characters translated from source encoding. The flex rules could not be written using literal characters “abc” and character classes “[a-z]”, which are host encoded. The solution was to use named entities “{LT}” corresponding to individual internal encoding characters. Each basic source character has its own entity ({a}, {b}, {SPACE}, ...), as do the translated extended characters {TRANSLATED} and the end of file {EOF}. The special flex entity <<EOF>> cannot be used, because it can only appear as a single entity in a rule.

7.3.2 Token buffering

The objects processed by pp_lex are not simple ucn characters, but entire ucn_token instances. The pp_lex_guts flex wrapper operates only on character values of the tokens, so it is necessary to pass the token another way. The tokens coming from pre_lex are simultaneously pushed in instance of ucn_token_buffer and their values are stored in internal flex buffer. The tokens are then extracted in the rules of pp_lex_guts and used to fill values of the created pp_tokens. The both actions have to be synchronized by calls to advance() on ucn_token_buffer to skip the unimportant tokens.

7.3.3 Include directive names

As stated in **[2.1/3]**, the process of dividing of source file’s characters into preprocessing tokens is context-dependent. This means, that when reading the *hchar* sequence (<file_name>), the content of the brackets is not scanned as ordinary tokens, but as a whole. Only if the form of the include directive is different from bracketed and double quoted, the text is scanned as normal tokens and expanded in later stages. Solution to this problem was to introduce a flag meaning “start of line outside macro expansion” into the otherwise stateless lexical analysis mechanism. The flag has to be passed directly into the call to lexer_parse() method. The flag is passed from the following stages, which are conscious of the state of the pending macro expansions.

7.4 Macro expansion

One of the most important preprocessor constructs are macros and macro expansion. The preprocessor distinguishes two types of macros,

- *object-like* macros, which have no parameters, (COUNT)
- and *function-like macros*, which can have zero or more parameters (max()).

To be able to obey rules concerning redefinition of macros **[16.3/1,2,3]**, it was necessary to define the equality operation on macros.

7.4.1 Expander

The expansion is performed by the `expander` class, wrapper for the stream of `pp_token`s coming from the lexical analysis. It has methods for returning expanded or non-expanded lines, of tokens, dependent on the state of the following stages of processing.

7.4.2 Operations

The Standard defines two operations in the expansion of macros, stringification and concatenation. The implementation has to distinguish more kinds, because the operations also influence the mode of processing of macro argument. The operations, represented as actions in `macro_item`, are

- `LITERAL` - copying the stored sequence of tokens literally,
- `EXPANSION` - expanding the value of the argument,
- `COPY` - copying the value of the argument literally,
- `STR` - stringifying the argument,
- `CONCAT` - concatenating the adjacent tokens in the stream.

7.4.2.1 Token concatenation

The concatenation of tokens by the `##` operator, as defined in **[16.3.3]**, is performed by the `concat` class. Its input are two `pp_tokens` to concatenate, the result is a `token_sequence`, containing either the concatenated token, or both the input tokens, if the concatenation would not result into a valid preprocessing token. The concatenation is performed on the token spellings. The resulting `ucn_string` passed to a special scanner (defined in concat_guts.l), which checks that the concatenation can be parsed as a correct preprocessing token.

7.4.2.2 Token stringification

The stringification of tokens by the `#` operator, as defined in **[16.3.2]**, is performed by the `stringifier` class. Its input is a sequence of `pp_token`s, whose spellings are returned as a single `pp_token` of type `TOK_STRING`. Characters, which are outside the basic source character set, are stringified as the universal-character-name escape sequences. The backslash and double quote characters are preceded with a backslash in the resulting string.

7.4.3 Object-like macros

Object-like macros are simple substitutions of name for a sequence of tokens. However, since the `##` operator can be used even for object like macros, the `macro_item` objects can be of type `LITERAL` or `CONCAT`. The expansion simply concatenates the tokens in the adjacent literals.

7.4.4 Function-like macros

Function-like macro expansion lists can contain references to parameters. The arguments substituted for parameters are expanded, unless they are a part of `##` or `#` operator, otherwise they are substituted literally.

7.4.5 Expanding macros

When a name of a currently defined object macro is encountered in the input, it is simply expanded by call to the `expand()` method. If the macro is function-line, it is checked, whether the next token is opening parenthesis and so the occurrence of the name is actually a macro call. Then the individual arguments are parsed into `macro_arguments` instance, issuing errors for unterminated argument list. The individual macro arguments are parsed inside the `macro_argument` class. They are separated by commas, but the commas inside nested parenthesis are not considered. The argument list is passed to the `expand()` method. For function-like macros, the method checks the parameter against the argument count. If they match, it evaluates all `macro_item`s contained in `macro_body`, resolving references to parameters to the passed arguments.

7.5 Preprocessing directives evaluation

When the preprocessor encounters a hash mark as a first token at the beginning of line, which is not a part of a macro call, it recognizes it as a preprocessing directive. Generally, the line containing the directive is not macro expanded, only in individual cases the rest of the directive after the directive name is expanded. The directives have to be parsed and evaluated. This is the task of the `evaluator` class. We now describe the behavior for individual directives.

7.5.1 Include directive

There are three kinds of include directive,

- `#include "file"`, which includes the file, searching in the current directory,
- `#include <file>`, which includes the file, searching also in the directories specified by the -I option,
- `#include` tokens, which expands the macros in tokens and tries to parse the directive again as one of the above.

Due to processing of the `#include` directive, the `evaluator` contains a stack of `unit_part` objects.

7.5.1.1 Unit part

Unit part represents a single part of the translation unit. It is either the top-level source file, or any of the included files. The `unit_part` class contains all entities necessary for processing the unit part which cannot be shared with the rest of the preprocessor. Among them are `pre_lex`, `pp_lex`, `line_control`, `expander` and `condition_stack`. A new `unit_part` is pushed onto the stack in `evaluator` for each included file

7.5.2 Conditional directives

Conditional directives include[2]

- `#if` condition,
- `#elif` condition,
- `#ifdef` testing defined macros,
- `#ifndef` testing undefined macros,
- `#else` branch,
- `#endif` ending the condition.

The conditional directives can cause parts of the source to be skipped **[16.1/6]**. The current state of skipping is controlled by a flag of the `evaluator`. Because the directives can nest, but do not overlap from nested included file into the parent, the state information is stored into unit part `condition_stack` instance.

7.5.2.1 Condition stack

The class `condition_stack` manages the conditional directives, represented by `condition`, which are active within a unit part. The `process()` method checks invalid nesting issues error messages. It has to be called not only for each encountered condition, but also at the end of the unit part, to check for unterminated conditions.

7.5.3 Other directives

7.5.3.1 Macro definitions

The #`define` directive performs call to `macro_storage` to define the macro parsed by `macro` class, issuing possible errors when the definition is invalid. If the macro was already defined, but the new and the old definitions are different according to **[16.3]**, an error report is issued. The `#undef` directive forces the `macro_storage` to remove the definition of the named macro.

[2]Directives `#if` and `#endif` are not covered in the implementation.

7.5.3.2 Line control

The `#line` directive is used to change the apparent file and line number of the tokens, starting from the next line in the input. The content of the directive is macro expanded before processing. It accepts new line number, which is a sequence of digits 1-32767 and optional string with the new filename. The change is performed via `line_control`, managing line numbers assigned to tokens created in `pp_lex`.

7.5.3.3 Miscellaneous

The `#error` directive simply issues an error message "user defined error", but without including the rest of the tokens into the error report. The `#pragma` directive supports only parameter "lestes", which causes the compiler to stop processing of the translation unit before ss2pi stage. Other `#pragma` directives are ignored, as is the empty `#` directive. However, it is an error, when some tokens are found trailing in directives, where they are not expected (`#else`, `#endif`).

7.6 Post-processing

After evaluation of directives, the stream of `pp_tokens` returned from `evaluator` has to be post-processed to obtain stream of `cpp_tokens`. The transformations include

- literal translation (`string_translator`),
- space removal (`space_remover`),
- string concatenation (`string_joiner`),
- token conversion (`preprocessor`).

The order of string concatenation and space removal is opposite to the one specified in the Standard, due to implementation reasons. The result is equivalent to that achieved through the proper order. Each of the mentioned classes, except from the last one, inherits from class `pp_filter`.

7.6.1 Preprocessing token filter

The `pp_filter` class is an interface, representing abstract filter of `pp_tokens`. The interface is congruent to the `ucn_filter`, only the processed element changes to `pp_token`.

7.6.2 Literal translation

In translation phase 5 **[2.1/5]**, the (wide) string and character literals are converted to the execution character set. This is performed by the `string_translator` class. The `\xhhh` and `\xooo` escape sequence values are passed directly into the execution character set, the overflow is not checked. The symbolic escape sequences and other characters are translated from the internal encoding into the target encoding. By now, the implementation only provides mapping from basic source characters into 7-bit ASCII character set.

7.6.3 Space removal

In translation phase 7 **[2.1/7]**, the white space characters become insignificant. The `space_remover` class eliminates all occurrences of `TOK_BLANK` and `TOK_LINE_END` from the stream. This transformation precedes the string concatenation, because that makes no difference in the result and the implementation of the latter becomes easier.

7.6.4 String concatenation

In translation phase 6 **[2.1/6]**, adjacent ordinary string literals and adjacent wide string literals are concatenated. The `string_joiner` also concatenates adjacent ordinary and wide string literals into a wide string literal, reporting an error. This behavior prevents from more serious errors that would occur in parser meeting two adjacent string literals.

7.6.5 Token conversion

The translation phase 7 **[2.1/7]** also specifies, that preprocessing tokens are converted into tokens. This is done in the `read()` method of the `preprocessor` class itself. Type of each `pp_token` is analyzed to eliminate invalid tokens ("`#`") and stray characters ("`\`") . Then the corresponding `cpp_token` is created, adding the proper origin to its `source_location`. Because the `cpp_token` types are dictated by the generated bison parser, the `cpp_token` types are not compatible with the `pp_token` types and a mapping must be provided.

7.6.5.1 Literals

If the converted preprocessing token is any kind of a literal (it has a type `TOK_*_LIT`), the `cpp_token` method `create_literal()` is called, passing it the appropriate `lex_*_literal` object created. For `TOK_NUMBER_LIT`, the special `preprocessor` method `classify_number()` attempts to parse preprocessing numbers, returning the appropriate `lex_integer_literal` or `lex_floating_literal` or `NULL` upon error.

7.7 Implementation specific issues

There are statements in the ISO standard, which are implementation defined. Each implementation shall document its behavior in these cases.

- The physical source file characters are interpreted as either 7-bit ASCII or UTF-8 and mapped to the internal representation of source character set.
- The whitespace characters are not retained, they are collapsed into a single one in the course of processing
- The backslash newline sequence can paste into the universal-character-name escape sequence.
- Nonempty source file ending in backslash newline sequence, or not ending with newline is rejected with an error.
- There are no additional members of execution character set supported.
- Only basic source characters are supported in header names.
- Value of multi-character literal is the value of the last character of the literal.
- If the value of character literal is out of range, it is truncated.
- The path for `#include` search is specified by -I option of the compiler.
- The character are combined into the header names via simple concatenation.
- Nesting limit is set to 16 to avoid infinite recursion.
- Only `#pragma lestes` is defined to switch off the ss2pi.
- Concatenation into universal-character-name escape sequence is not recognized.
- The quote or double quote character falling into other token category issues an error message about unterminated literal, but the token is still created.
- Escape sequences are not allowed in first type of header name.
- Invalid escape sequences cause error to be issued.
- If the third type of include does not match, error is issued.
- Empty macro argument is allowed, behave as if no tokens were there.
- Sequences in parameters resembling directives are completely legal.
- Stringification of all tokens is escaped to be used
- Concatenation by `##` not resulting into token is error.
- Directive `#line` with number 0 or greater than 32767 is an error.
- Directive `#line` with unrecognized content is an error.

Chapter 8

Syntactic Analysis and Passing Data to Semantic Analysis

8.1 Introduction

Compared to other languages, analyzing C++ is much more complicated because the language is ambiguous. The same input can have more than one interpretation and the correct one can be usually recognized using type information. This means that the analyzer can inquire whether the identifier denotes a type or not **[6.8/3]**. In some situations, even the type information is not enough to resolve the ambiguity. Then the Standard states which of the possibilities is to be considered.

The following example demonstrates many problems that the analyzer has to challenge.

```
int i, j, k, l;
class C { /* ... */ };
int f( int (i) );           // (1) function taking int argument
int f( int (C) );           // (2) function taking function argument
int g() {
  C (i)++;                  // (3) expression-statement
  C (i);                    // (4) declaration-statement
  C (j) = 5, (k) = 8, l++;  // (5) expression-statement
  C (j) = 5, (k) = 8, l;    // (6) declaration-statement
}
```

Ambiguity on lines `(1)` and `(2)` between argument of type `int` with redundant parentheses around its name `i`, and argument of type "function taking `C` and returning `int`". This particular problem can be solved using type information only. Nevertheless, the ambiguity resolution mechanism in Lestes (dubbed disambiguation) does not explicitly use the type information to commit to the right interpretation.

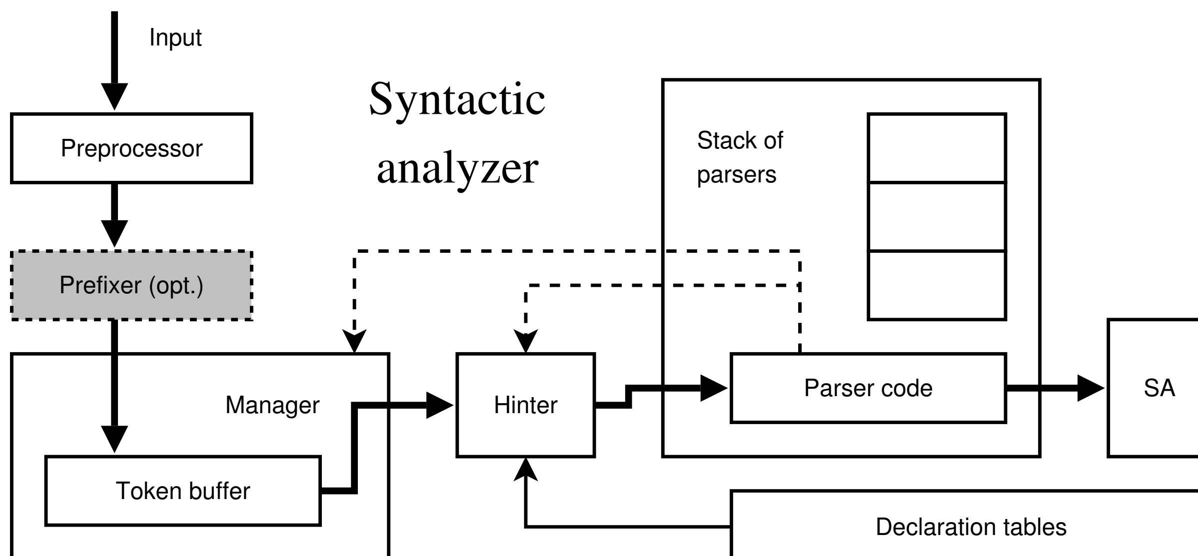
Line `(4)` contains rather simple, yet ambiguous code. It can be an expression in which a temporary object of type `C` is constructed from `i`, or a declaration of object `i` of type `C` with redundant parentheses (it hides the global integer with the same name). In this case the Standard explicitly states that declaration has precedence over expression **[6.8/1]**. However, if ordinary LALR(1) automaton was used, it would not know which grammar rule is to be used after reading the closing parenthesis. Sometimes – as seen on line `(3)` – the type of token that follows ("look-ahead") could help. However, exploiting this tiny opportunity would be extremely difficult.

Examples on lines `(5)` and `(6)` display that the token which eventually determines the final resolution can be located much further in the token stream. Writing grammar that would cover *and* solve these ambiguities would not be readable (if possible at all). The former line contains three expressions (assignment to temporary of type `C`, assignment to integer `k`, and post-incrementing of `l`) concatenated using comma (`,`) operator; the latter declares `j`, `k`, and `l` constructed using assignment syntax. Note that "capabilities" of the types in question cannot be taken into consideration **[6.8./3]** when deciding which of the interpretations are viable.

Another approach to these issues could be creating grammar that accepts a super-set of the language that would not distinguish between the possible alternatives. These would have to be processed and decided upon later, in semantic analysis. We did not choose this path because the resulting grammar would significantly differ from the one found in the Standard, and processing the language super-set later (in SA) would not be trivial. Above all, we found a way to do this purely on the syntax level, just as the standard suggests **[6.8/3]**. This is not to say that the analyzer does not use type information found in declaration tables (filled exclusively by SA).

This chapter covers the details of the implementation and assumes that the reader is familiar with bison development.

8.2 Overview



The syntactic analyzer in Lestes has two key elements. The disambiguation manager buffers input and passes the tokens through hinter to the parsers. They are the second key element. Hinter is not that relevant to the actual syntactic analysis, yet the type information it supplies is needed.

The parsers are bison-generated LALR(1) automatons. They are re-entrant to allow for recursive invocation. Logically, there are several stacked parsers, but they are sharing one copy of the code, as they are generated from a single bison grammar file. For simplicity, the code generally does not detect which depth it runs at.

The whole analyzer (manager, hinter, parser) is reentrant: at any time a new analyzer can be started.

8.3 Disambiguation Manager

The main purpose of the manager is to supervise the process of disambiguation. The disambiguation itself is controlled by the parsers - they call the manager because they “know” where, and what kind of disambiguation needs to be run. The manager is responsible for manipulating the input token stream as requested by the parsers.

The manager also does *packing*, a method for delaying analysis of some chunks from the source. Again, packing itself is controlled from the parsers yet performed by the manager. For example, when the parser expects function body to follow, it makes the manager pack it and reads it as one token which contains the original token sequence inside. The body is extracted verbatim when its analysis can be conducted – usually when the tables are filled so that the hinter can correctly supply type information.

8.3.1 Disambiguation

The Standard mandates that the disambiguation precedes parsing **[6.8/3].** That implies the analyzer has to break out from parsing and process incoming token sequence sideways to find out the possible interpretations. If there is more than one of them, the analyzer has to choose the right one according to the Standard. It usually states which of the possibilities is to be considered. Sometimes, the Standard explicitly says that one of the possibilities takes precedence (e.g. declaration-statement before expression-statement **[6.8/1]**). In other cases, these precedences can be worked out.

Syntactic analysis builds on the assumption that ambiguity resolution directives are formulated as precedences.

Whenever parser reaches a state requiring disambiguation, it is already known which two (or more) ambiguous constructs are allowed to follow (declaration-statement and expression-statement from the example above). For each such construct a new “one-purpose” parser is started at the very position in the source token stream. Each construct has its own specialized parser.

The sole purpose of the newly started parser – which runs while the main one is paused – is to find out whether the following token sequence can match specific language construct. During this trial pass we do not modify identifier (declaration) tables and the only outcome of the pass is a logical value: Whether parsing succeeded (the construct matches), or not.

This way, all of the constructs in question are tried, starting at the same position in source. As all disambiguation rules are precedence-based we can run the specific parsers in order dictated by these rules and accept the first successful try as result of the disambiguation. When the result is computed, the main parser continues knowing which of the paths to choose. Technical details of implementing this within a bison-generated parser will be discussed later in implementation section.

8.3.1.1 Details

The manager has to buffer input tokens, because after the specialized parser returns, the token stream has to be rewound back to the position where it was before starting the trial parser. Therefore a buffer is needed. The manager keeps a buffer of tokens as long as a disambiguation is conducted. In other cases, it keeps just the last token (needed when backing up the look-ahead token).

The newly started one-purpose parser needs a starting token (see 8.5) to know which construct to try processing. This is simply inserted just before the token that would have to be read. This way, it is sufficient just to call the paring method (`parser::parse()`) and it will see the START token as the first token on input. Tokens that follow will not be changed.

There are two possible outcomes of running the special parser.

- parsing failed. This means that the try failed and the START token has to be removed. The disambiguation is said to *rollback*.
- parsing succeeded. The START token is replaced with a PAD token that will be seen by the upper parser. The disambiguation is said to *commit*. The PAD token then indicates the choice that the upper-level parser should make. The decision depends on the value of the start token.

Note that it is not the manager who decides whether the parsing succeeded or not. It is the caller of the `parse()` method. The manager just inserts a token to the stream and iterates over the buffer to feed the parser.

8.3.1.2 Nested disambiguation

Important thing to notice is that disambiguations can nest. In other words, while resolving an ambiguity, another ambiguity might arise. The concept is the same, however. We try parsing the possible sub-constructs while the parser which was itself parsing some construct is not running. After resolving the sub-ambiguity (accepting the first matching successful try) the parser knows how to carry on.

However, a problem arises when a sub-ambiguity is committed, but its parent ambiguity fails later. The pad tokens inserted into the buffer need to be removed. In these situations, the manager uses semantics of nested transactions. Whenever a transaction (process of disambiguation) rolls back, all sub-transactions that committed while it was running must be rolled back too. This is straightforward to implement: when a disambiguation commits a PAD token into the buffer a “compensating transaction” (removal from the buffer) is passed to parent transaction, together with all compensating transaction received from successfully finished children.

The algorithm ensures that whenever a disambiguation attempt fails, all left-over PAD tokens from any and all sub-attempts are removed.

8.3.1.3 Delayed analysis

Some parts of the language cannot be parsed immediately, because they are declared in a scope that is not finished, still they need to “see” all the declaration from that scope, possibly ones that are yet to be read from the input. Great example is a method body defined in the class body.

```
int i;       // (1)
class c {
  int f() {
    i = 5;   // (2), refers to this->i declared below!
  }
  int i;     // (3)
};
```

As stated in [3.4.1/8], the `i` on line `(2)` refers to `i` on line `(3)`, not the global one from line `(1)`.

We solve this problem by storing the whole method body unparsed, and resorting to parse it when possible, presumably at the end of class declaration. This approach is called *packing*. Packing is requested by the parser and happens in manager. Depending on the kind of packing that needs to be performed, the manager reads tokens until the terminating one is found, and removes the sequence from its buffer. This sequence is stored into `bison_id_token` that can be shifted as a single token when control returns to the parser that initiated the packing.

In some rare cases, packing is used during disambiguation, so it must be compensated when a parent disambiguation fails. This is just another kind of compensating transaction – one that unpacks the pack back to the buffer.

8.3.1.4 Error recovery

While error recovery was not a concern when the analyzer was designed, it is rather easy to implement it at some points. At points where the disambiguation is run.

When a state requiring disambiguation is reached and all of the tries failed, we usually know what does the incoming stream of tokens look like. This enables us to decide what tokens we have to search for when finding a place to continue the analysis.

The packing process can be abused to do this. The manager is called to pack a sequence of tokens. The packing method finds the skeletal token and packs the ones between. Instead of a PAD token that tells which of the possible ambiguous constructs to choose, a PACK token is committed. The rule containing the disambiguation run has to be prepared for this. By shifting over the pack token, it in effect skips over the erroneous input and tries to recover later.

Currently, this is implemented in the simple cases. Ill-formed conditions are skipped until the closing parenthesis is read. We are also jump over incorrect statements at function scope. In this case, we continue after first semicolon (`;`), or before braces (`{`, `}`) or keywords `do`, `for`, `while`, etc.

Given the current design and implementation, more cannot be done to recover from ill-formed input. When the main parser stops with syntax error there is no way to restart it at the syntactic context where it left off. Bison error recovery features were not considered as they might interact with the disambiguation.

Nevertheless, as function bodies are always processed in separate analyzer, it is possible to “recover” by skipping to the end of the function body and move over to the next one. The only benefit is that the user might be informed about error in a different function.

8.3.2 Internals

The following text briefly describes interface used by the parsers. Note that there is a stack of re-entered managers, each with its own parser and hinter. Although these functions are static method of `manager` class, they are just forwards to the manager at the top of the stack. This is implied by the fact that the actions in parsers are the same in all instances of the analyzer and *always* want to call the current manager (the one on the top of the stack).

8.3.2.1 Token reading

The manager provides an enhanced version of interface required by a bison-generated automaton. In addition to the necessary `yylex()` function, similar functions for looking back and forward are provided.

- `yylex()` — returns `bison_token` from the stream and moves the internal pointer to the next one
- `peek()` — returns `bison_token` from the stream, but does not move the pointer; used when code other than the automaton want to know what kind of token is ahead
- `yylex_prev()` — returns token that was last result value
- `back_up()` — moves the internal pointer one step backwards; used when the automaton has read the lookahead token, but a disambiguation has to take place *before* it

8.3.2.2 Disambiguation control

Invoked from within the disambiguation rules, the following functions insert/delete PAD/START tokens from/to the input stream buffer.

- `start()` — inserts given START token into the token stream; it will be returned by next invocation of `yylex()`; usually a new parser is starter immediately
- `commit()` — rewinds input to the latest START token and overwrites it with provided PAD token, which will be returned by `yylex()`; used to confirm a resolution of an ambiguity
- `rollback()` — rewinds input to the latest START token, removes it; also removed are all PAD tokens from successful sub-commits
- `in_disambiguation()` — returns `true` when a disambiguation is running, i.e. there is at least one unconfirmed/not rolled back START token
- `pack()` — given wanted type of pack token, packs the input that follows; `yylex()` is used to read it
- `unpack()` — unpacks token on input; the following token must be a pack token; used rarely, and is actually unnecessary

8.3.2.3 Spawning new analyzers

- `spawn()` — spawns a whole new manager-parser-hinter combo; a pack token must be given: its contents are extracted and will be returned by `yylex()` and equivalents; currently, only used to parse method bodies
- `close()` — shuts down the analyzer

8.4 Hinter

The hinter processes all tokens when they leave the manager, but before they are passed to the parser. For identifiers, it searches the tables for declarations with the same name, following the rules described by the Standard as Name lookup **[3.4]**. The lookup result determines type of the token that reaches the parser.

The name hinter comes from the fact that it actually gives hints to the parser. The word `HINT` is being used throughout the code, "type information" is used in this documentation however.

8.4.1 Operation

Hinter runs the search for every `bison_token` with `TOK_IDENT` type and creates `bison_id_token` with type corresponding to the result of the search. Other tokens are left untouched. The grammar recognizes the following kinds of identifier tokens. Non-type identifiers:

- `UNKNOWN_ID` — unknown identifier
- `NONTYPE` — name of an object, or a function
- `TEMPL_NONTYPE` — name of templated function

Type identifiers:

- `CLASS_NAME_ID` — name of a class, or name of a typedef of a class type
- `TYPEDEF_NAME_ID` — typedef of non-class type
- `ENUM_NAME_ID` — name of an enumeration type
- `NMSPC_NAME` — namespace name
- `TEMPL_TYPE` — name of a class template

As requested by the standard [6.8/3], the actual kind of identifier is generally not being used. The important part of the information is whether the identifier denotes a type or not. Most of the time, a particular rule matches any of the tokens from the group.

In its current implementation, the hinter does not recognize template names, as corresponding SS structures are not implemented.

8.4.2 Interaction with other parts

Hinter must be tightly knit to both the semantic analysis and the parsers. The former is required because the lookup performed by the hinter must be run in correct scope. Qualified names, or elaborated specifiers imply the latter, as these influence parameters of the lookup process.

Set of found declarations is then bound to the identifier token and may be used if needed. Note that the search takes place at all times – even when the results are not actually required.

8.4.2.1 Qualification

Behavior of the lookup procedure varies quite much when looking for different parts of a qualified identifier. In `A::B::C` different rules apply to all three components of the name **[3.4.3]**.

Qualification (application of the `::` operator) is handled in the parser. The hinter is informed using its `qual_scope_set` method.

8.4.2.2 Elaborated specifiers

Elaborated specifiers are names prefixed by one of the following keywords: `class`, `struct`, `enum`. If one of these prefixes is used before an identifier, the lookup has to ignore non-type names **[3.4.4]**, for example. When the parser shifts over one of the keywords above, it calls the hinter's `elab_spec_set` method.

8.4.2.3 Scope

The scope where to perform the lookup is provided by SA context manager (see page 79).

8.4.2.4 Set of found declarations

The set of declarations that have been found by hinter when doing lookup for the identifier are then stored into the `bison_id_token` that is passed to the parser, which wraps it in `as_identifier`. This set of declarations is extracted later in SA. This provides a slight speed-up, as in many cases SA does not have to perform the same search again. When set of declarations for the name in question are needed, information found in the hinter is easily accessible.

8.4.3 Prefixer

Prefixer is a token filter used to debug the grammar, the lookup code, and/or the hinter. It operates between the preprocessor and the hinter. When turned on, it allows developers to specify value that should be used as if it was computed by the hinter. Identifiers that are prefixed by another identifier with special name (e.g. "`_hint_class`") is tagged with "user hint".

Actually it is the hinter that processes these tags and decides what to do. Two modes of operation are implemented at the moment:

- `HINTER_OFF` — normal hinting is completely turned off and the user-provided hint is trusted. A hint *must* be present before every identifier. However, this is only useful to debug the grammar, as SA expects to find the set of found declarations which is not (and cannot be) supplied.

- `HINTER_CHECK` — hinter works as usual; when the user specifies a hint, it is compared to the computed one and if they do not agree an error is printed.

Prefixes also provides ways to hint the disambiguation process. For full list of prefixes see `prefixes.cc`.

8.5 Parsers

Logically, there is exactly one *main* parser (accepting a complete source program, a function body, ...) which calls the special parsers used to parse for purposes of disambiguation. Each of these parsers recognizes its part of the language, each needs a separate grammar. There is only one grammar file, however.

By prefixing each of these grammars with a unique terminal (token), we can merge all the grammars into one. Yet the parsers are virtually separate. After reading the very first token, it is clear what grammar from the set is about to be parsed. We call these first tokens START tokens.

One huge grammar has two advantages over creating several grammars in separate files: The algorithm used by bison merges "similar" states, although they are are reachable through different START tokens. This is actually a win, as it decreases size of the generated automaton which results in less C++ code. The other, more important, advantage is the ability to reuse rules from the main grammar in the specialized grammars. This makes writing the specialized parser rules trivial: START token followed by the construct to be tried and an action that terminates the parser with success.

Apart from containing the START tokens and rules for the special parsers, the grammar very closely resembles the one from the standard.

8.5.1 Disambiguation rules

Rules that actually run the disambiguation are always "empty" – they reduce to empty sequence. They are inserted before constructs that are ambiguous. The right places are easily identified in bison's output file, which contains detailed description of the generated automaton.

The code that is run when the disambiguation rule is reduced by, always follows the same pattern:

- if the following token is a PAD token, it is a result of this same disambiguation[1], *do nothing*; otherwise
- for every possible construct that is allowed to follow, *try to parse* it with specialized parser; if it *succeeded, stop* the loop, *otherwise try the next* construct

Where "try to parse..." means

1. run `manager::start()` with START token of the construct; the token is inserted into the buffer and will be returned next time any parser requests a token from the manager
2. run `parser::parse()`; this re-enters the parser, the specialised parser is determined by the START token specified in previous step
3. check result of the recursively called parser
   (a) if the parsing succeeded, commit the disambiguation with appropriate PAD token by calling `manager::commit()`; this rewinds the token stream position to where is was before calling `start()`
   (b) if the parsing failed, call `manager::rollback()`; this removes the START token, as the tried construct was not possible at this place; the position in token stream is rewound the same way as int the successful case

If all the tries failed, a syntax error was found. In places where we can guess where the ill-formed sub-construct ends, we can pack and try to resume parsing after it. The erroneous sequence is packed into a different PAD token that is recognized later.

After the ambiguity is resolved, the automaton must expect the PAD tokens. These usually follow the nonterminal representing (and performing) the disambiguation effort. In the grammar, this looks like:

```
statement: disambiguate TOK_PAD_DECLARATION declaration_statement ;
statement: disambiguate TOK_PAD_EXPRESSION expression_statement ;
statement: disambiguate TOK_PAD_ERROR ;
```

Where `disambiguation` follows the algorithm outlined above. The PAD tokens committed by the disambiguation actually makes the decision that is then used by the upper-level parser. Note that nested ambiguities are handled the same way.

When all sub-ambiguities are resolved the main parser is said to *parse for real*. Resolutions (PAD tokens) of all the upcoming ambiguities of currently processed sub-construct are already committed to the token stream.

[1]This happens when parsing for real and sub-disambiguation is run, but its resolution (PAD token) is already present.

8.5.2 Look-ahead token issues

It is ideal for the disambiguation rule to be reduced in a state that does not need value of the look-ahead token. However, this is not always possible, if we want to keep the grammar readable and as close to the original one from the standard. In these cases, we have to be careful and make sure that the automaton does what we want it to.

For example, within declarators, ambiguities arise between constructs in parentheses (`(,)`). In some states, we usually do not need to run disambiguation when the following token is different from the opening parenthesis. What is more, not adding the disambiguation to rules that do not need it keeps the grammar maintainable. However, this might get a little tricky.

Imagine the following grammar rules (the real situation is simplified for explanatory purposes):

```
direct_abst_declarator: '[' constant-expression ']' ;
direct_abst_declarator: disambiguate_in_ad TOK_PAD_AD '(' abst_declarator ')' ;
direct_abst_declarator: disambiguate_in_ad TOK_PAD_PDC parameter_declaration_clause ;
parameter_declaration_clause: '(' ...
```

Here, empty abstract declarator enclosed in redundant parenthesis conflicts with parameter declaration clause with no parameters[2].

When the automaton reaches the state that covers start of this rule, it has to read the look-ahead token to decide whether to reduce by the empty disambiguation rule (its action would run the disambiguation itself), or to shift the bracket (`[`). However, the actual disambiguation is only run by the automaton when one of the PAD tokens follow! This obviously does not work, as the PAD token is not committed into the token stream yet.

What makes this problem hard to track is that sometimes, the bison algorithm merges reduction by the disambiguation rule into "default reduce" action, an analogy to the else branch in if-then-else construct. When this happens, the disambiguation appears to work, but further changes, that apparently do not relate to this part of the grammar, make the generator produce automaton that reduces by the disambiguation rule only when one of the PAD tokens follow.

The situation is saved by adding rule that explicitly contains the right token:

```
direct_abst_declarator: disambiguate_in_ad '('
```

This makes the automaton reduce using the disambiguation rule when it sees the parenthesis ahead. On the other hand, when the automaton reduces by this artificial rule, something want wrong, as the disambiguation was supposed to commit a pad token. In Lestes, an assertion is put into action belonging to this rule.

There is one more concern to this approach: We have to make the automaton re-read the look-ahead token, as its value was changed by the disambiguation. This is accomplished in the disambiguation code by assigning `YYEMPTY` to the `yychar` variable that holds the parser's look-ahead token. This makes the machine re-read the look-ahead token when it returns from the disambiguation state, despite the fact that a decision was already based on the previous value.

When the parser read the look-ahead token to decide whether to run a disambiguation, the manager must be told to move the buffer position one step backwards so that the START token is inserted *before* the just read one. `manager::back_up()` is designed to achieve this.

"Entry" to every disambiguation rule is guarded by another empty rule that – in its action – checks whether the look-ahead token is read. There are two rules (`lookahead_assert_empty`, and `lookahead_assert_non_empty`) which assert that the look-ahead was, or was not read to make the decision to enter the state, respectively.

8.5.2.1 Disambiguation at function scope

Disambiguation that takes place at function scope deserves special treatment. It is used to resolve ambiguities between expression-statement and declaration-statement (for an example, see section 8.1 on page 63). Apart from these, function scope can also contain constructs like `if`, or `while` that are not ambiguous.

The first proposal is to run the disambiguation only when the look-ahead token is one of those, that an expression or a statement can start with (known as the FIRST set), and leave this decision up to the automaton. However, this implies that all tokens from the START sets of both the expression and declaration statements need to be specified after the disambiguation nonterminal – to avoid problems described above. Maintaining the sets by hand is error prone, as the sets are relatively large[3].

---

[2] In this case, abstract declarator has precedence over parameter-declaration clause and is tried first.

[3] Actually, the first version of this type of disambiguation suffered from the very problem from previous section, because tokens from the FIRST sets were not used. However, for a long time bison generated automaton, that had the right reductions in hidden in default reduce actions and the disambiguation worked. Further changes required by SA unveiled this mistake by chance.

Currently, this disambiguation is run before *every* statement, without the look-ahead being read. Then the code in the action for the disambiguation rule decides whether to actually try to disambiguate, or not. And unlike the automaton, it chooses when *not* to disambiguate, as the set of tokens not requiring action is easy to assemble. It contains keywords like if, for, catch, else, etc. and both braces ({,}). The tiny disadvantage is that the nonterminal that runs this code has to be added to much more places. Nevertheless, this solution proved to be much more robust.

8.5.3 Semantic values

Besides resolving ambiguities, the parser also build the AS tree that is passed for semantic analysis at specific times. When building the tree, intermediate values are stored on the stack of the generated automaton. These are called semantic values. Bison implementation statically allocates the semantic values stack as array of object of single type. Bison translates occurrences of $N (where N is a number) in actions to code that accesses the different types of semantic values using member access (.) notation, as if the type was a union.

In C++ we can exploit this to call specific functions on our specially crafted class objects. The semval class that is used as semantic value type contains just a pointer to as_base – the ancestor of all AS structures – and a templated select function that returns instance of semref class template that is instantiated with requested type. Intermediate object of type semref<T> tries to dynamically cast the as_base pointer to type ptr<T> when dereferenced. T has to be derived from the base, of course.

The implementation checks for the following:

- when dereferencing semref<T> the actual value stored is of type derived from T – done at run-time,
- assigning ptr<T> to semref<S> is only allowed when T is derived from S – done at compile time,
- assigning semref<A> to semref<B> is always allowed – this happens when overwriting left-over values on the stack.
- assigning null to any semref object is erroneous – using null pointers renders some of the type checking above partially useless, because their dynamic type cannot be verified; when a null value needs to be stored, a wrapper is needed (see as_other.lsd)

However fragile this might seem, the compile-time check together with strictly typing the intermediate values using %type in bison make the result work without any problems whatsoever.

8.6 Token types

Syntactic analysis has its own set of tokens. They are described in section 3.7 on page 35.

8.7 Passing Data to SA

SA is called by the parser at appropriate places from the rule actions. It is given needed AS structures, which are being constructed by the parser.

To reduce size of the parser code, SA methods are always called. Even when disambiguating, when the declaration tables should not be changed. The methods themselves must check back at the manager to find out whether we are only disambiguating, or parsing for real.

8.8 To Be Implemented

- As *error recovery* was added to the analyzer as an afterthought, in needs to be redesigned. This might involve deeper changes.
- To fully support all C++ features, the *hinter* will need to handle typename names as well as templates.
- The parsers are ready for *constructors* recognition, but this will have to be connected to relevant parts of SA to work.
- At the moment, the analyzer only *packs* function bodies. More constructs of the language need delayed analysis. This is straightforward to implement.

Chapter 9

Lookup

Lookup is the process of identifying the meaning associated with a name. The identification is done using parametrized traversal of the ss_decl_seq structures.

The lookup process is a fundamental part of the compiler. It is used in the hinter (cf. section 8.4), during the processing of declarations (cf. e.g. subsubsection 10.5.2.4).

There are various needs that the lookup implementation has to satisfy. The interface is provided in a manner that allows to fulfill the various needs, yet permitting to maintain the lookup processor relatively simple.

9.1 Lookup API

There is only one API that lookup mechanism in Lestes compiler is providing to the "outside world". This interface is represented by singleton class, called lu_lookup and the invocation of the lookup process is performed by calling the method main() of this singleton with proper parameters, i.e.

```
lu_lookup::instance()->main(seq,par)
```

In this call, the seq parameter is a pointer to decl_seq (see explanation in chapter concerining Semantic Structures for explanation of the term decl_seq) in which the lookup is to be performed. This decl_seq can be of any kind - it can "originate" from class, namespace, compound, etc. The par parameter specifies the parameters of the lookup, which among other things specify the behavior when handling using directives and using declarations. The behavior with respect to the using directives is prescribed by Standard and differs from situation to situation, in which the lookup is performed and what declaration is being looked up. On the other hand, handling of using declarations is almost everytime prescribed to be resolved, and this is specified by the UDECL_RESOLVE flag.

The next important thing that has to be specified looking up a given identifier, is whether the lookup should ascend to parent decl_seqs recursively or the lookup process should stop after the search just in the given decl_seq is completed. Searching just in one decl_seq is useful for example when looking up declarations for builtin operators, as prescribed by **[13.6]**, as the declarations for builtin operators specified there have their own decl_seq, where all appropriate builtin operator declarations are stored, and searching in other decl_seqs doesn't make sense.

Maybe the most important of the whole lookup API is filter. This is where all the matching, according to the name, the type of declaration, visibility, etc., is done. If there is need to match against multiple criteria simultaneously, it is possible to use more than one filter, which are then added to structure lu_multi_filter (representing multiple filters) by it's method add_filter().

9.1.1 Example

For the most commonly used types of lookups, there are files implementing them, allowing programmer just to use predefined interface, without having to set up the filters and the whole lookup environment. These files are located in the sem/ directory and starting with li_ prefix (which stands for Lookup Interface).

In the following example, we will show how lookup for the function declaration of a function of given name, is done (this lookup ascends into the parent scopes, as can be seen from the specified parameters)

```
ptr < lu_name_filter > nf = lu_name_filter::create(name);
ptr < lu_func_cont_filter > ff = lu_func_cont_filter::instance();
ptr < lu_multi_filter > mf = lu_multi_filter::create();
```

```
mf->add_filter(ff);
mf->add_filter(nf);
ptr < lu_params > params = lu_params::create(
     lu_params::UDIR_IGNORE,
     lu_params::UDECL_IGNORE,
     lu_params::SEARCH_PARENTS,
     ss_declaration_time::create(name->location_get()->order_get()), mf);
ptr< ::lestes::std::set< srp< ss_declaration > > > res =
                  lu_lookup::instance()->main(scope, params);
ptr< ::lestes::std::set< srp< ss_function_declaration > > > res_f =
           ::lestes::std::set< srp< ss_function_declaration > >::create();
::lestes::std::set< srp< ss_declaration > >::iterator it;
for(it = res->begin(); it != res->end(); it++){
     res_f->insert((*it).dncast<ss_function_declaration>());
}
return res_f;
```

9.1.2 Lookup Parameters in More Detail

A relatively small set of parameters direct the lookup process. The parameters specify the way using directives and declarations will be handled, whether the parents will be considered, and what declarations shall be selected. The parameters are passed to the lookup as an instance of the class `lu_params`.

The parameters are divided into three groups - handling of user directives, handling of type and parent search. The possible values for each of these three groups is described below:

1. handling of using directives.
   (a) `UDIR_IGNORE` - this instructs lookup not to consider using directives at any phase of the lookup. This is used for example in argument-dependent lookup.
   (b) `UDIR_ALWAYS` - this instructs lookup to consider using directives even when looked up identifier was found. This is used for example in unqualified lookup.
   (c) `UDIR_FALLBACK` - this instructs lookup to consider using directives only when looked up identifier was not found. This is used for example in qualified lookup.
2. handling of using declarations
   (a) `UDECL_IGNORE` - this instructs lookup not to process instances of `ss_using_declaration`, representing the using declaration on the SS level, at all.
   (b) `UDECL_THROUGH` - this instructs lookup to process instances of `ss_using_declaration` as other kinds of declarations. This means that instances of `ss_using_declaration` are not resolved, but processed verbatim.
   (c) `UDECL_RESOLVE` - this instructs lookup to resolve instances of ss_using_declaration and process their targets.
3. parent search type. This field can have only two values, `SEARCH_PARENTS` or `SKIP_PARENTS`, meaning of which is clear.

9.1.3 Filters

There are some predefined filters for the most common cases, in the file `lu_filter.lsd`. There are filters for namespace declarations, struct declarations, name declarations, etc. Declaring the "rules" for the filter is as simple as specifing the visitor method, which has to be called for filter to succeed.

The filters can be merged together (in logical conjunction), using the `lu_multi_filter`, as was shown in the example above.

9.1.4 Return Value

The `main()` lookup routine returns set of pointers to `ss_declaration`, which contains those declarations which were found according to the parameters passed to lookup.

9.1.5 Overview of Lookup Internals

The lookup algorithm can be divided into two main parts: the scope processor and the parents processor. The scope processor walks through all the declarations contained in that scope and all scopes nominated by using directives (while respecting the `UDIR_*` mode requested by the parameters). Each declaration is examined whether it is declared before the time for which the lookup is performed.[1] The using declarations are handled as specified by the `UDECL_*` enumerator passed as the parameter. The lookup filter is applied to the declaration and the declaration is added if the filter responds with `FR_YES` or `FR_YES_CONT`. The search stops on the `FR_YES` result and continues on `FR_NO` and `FR_YES_CONT`.

9.1.5.1 Filter Internals

Each filter is a visitor over the ss_declaration class hierarchy, to make the handling of declarations relatively easy. Most filters do distinguish between the kind of declaration expressed by the dynamic type of the `ss_declaration` descendant. While the filter itself is a descendant of a ss_declaration visitor this eliminates the need to have two classes for the most common kind of filters.

[1] This treatment permits the lookup to be performed with identical results at the point a name is encountered as well at any later time.

Chapter 10

Structure Analysis

10.1 SA Introduction

10.1.1 Transformation and Analysis of AS Structures

The parser constructs instance of AS-intercode classes. Every AS-intercode class models a language construct as viewed from the standpoint of an abstract syntax grammar. Even the name, AS, was chosen so as to reflect this fact.

The AS-intercode is not suitable for code generation. It does not distinguish between user and builtin operators; it does not represent implicit conversions, and it has many other properties making it hard to generate some pseudocode directly from As structures.

It is therefore necessary to transform AS structures to some other, more easily manageable structures. This transformation creates structures, that are more convenient in terms of name lookup, pseudoinstruction generation, data layout determining, type checking, &c.

These new structures constitute the next intercode layer named SS-intercode, i.e. Semantic Structure intercode. The process of transforming AS-intercode to the SS-intercode is collectively named SA – Structural Analysis.

10.1.1.1 Parts of SA

SA can be divided into three main parts. These parts collaborate to provide all the necessary information to the later stages of the translation.

The first part consists of declaration handling. This part is responsible for creation and filling of name tables used in the lookup process. It is also responsible for associating the type information to each of the declarations, for resolving whether a declaration is valid in a given context, for finding the declarations (if any) that are embellished by a given declaration, and certain other minor tasks. It is discussed in detail in section **10.5.**

The second part consists of statement handling. The responsibility of this part includes: the creation of corresponding statement structures for any statement, the handling of compound statements including the involved context switching, the enforcement of non-redeclarability rules concerning object names declared in the condition or init-statement of conditional and iteration constructs, and other minor tasks. This part is discussed in detail in section 10.4.

The third part consists of expression transformation. The expression transformation involves determining the type of subexpressions, finding implicit conversion sequences, overload resolution, sequence point establishing and other minor tasks. This part is discussed in detail in sections 10.6 and 10.7.

The first two parts are called into by the parser, because it is necessary to perform their actions early, so as to provide contextual information to the lookup hinter (and hence to the parser), and subsequent SA actions.

10.1.1.2 Common Terms of SA

The Standard provides a convenient device for describing semantics of C++ program, called *sequence point.* It is defined as a point in time when all sideeffects of the previous evaluations are complete and no sideeffects of the following evaluations have begun. The Standard also specifies places where these sequence points occur. Let's take the expression statement

```
a,b;
```

as the basis of our discussion. The Standard mandates that there are sequence points at the end of evaluation of a full expression. For convenience we can assume a sequence point at the beginning of such full expression evaluation as well.[1]

The Standard also mandates a sequence point at the comma,[2]

further requiring that the left subexpression is evaluated before the sequence point and the right subexpression evaluated after it.

Let's consider another expression statement [3]:

```
(a,b) + (c,d);
```

This expression statement provides four sequence points (one coming from the preceding statement). Firstly, there are the two boundary sequence point α and β, αcoming before the full expression and β coming after it. Then there are two other sequence points γ and δcoming from the first and second comma, respectively. The order of evaluation of the subexpressions of the + operator is not specified.

The decision whether to evaluate the left subexpression first or second is a matter of optimizing the resulting machine code. It should be therefore postponed to the point where architecturally dependent information used for the selection is available: to the backend.

The information that can be obtained is the information whether the order of evaluation is or is not prescribed. If the order of any two evaluations is prescribed, a sequence point representation is made to separate them. The evaluations (represented by instances of *ss_expression* derived classes) hold pointers to the sequence point that occurs before them and the sequence points that occur after them. These points are known as "previous sequence point" and "next sequence point", or colloquially *psp* and *nsp*.

The sequence points themselves have to contain similar information. Let's define some convenient notation for *psp's* and *nsp's*. Now, **psp(x)** would be defined as the sequence point, that is pointed to by the x's *psp* field. Similarly for **nsp(x)**.

In our example with the operator +**,** the following equations can be made to describe the example

psp(a) = α nsp(a) = γ
psp(b) = β nsp(b) = β
psp(c) = α nsp(c) = δ
psp(d) = δ nsp(d) = β
psp(γ) = α nsp(γ) = β
psp(δ) = α nsp(δ) = β
psp(β) = α

10.1.1.3 Level of a Sequence Point

It can be seen that any valid ordering can be obtained by topological ordering of the psp-nsp graph (either the psp or the nsp edges (but not both) would be considered in the opposite direction for the purposes of the ordering). It can be also seen that there are topological orderings that do not correspond to any valid ordering of the operations. To fully represent the semantics of the source expression, and additional information, called level of a sequence point, is needed. The *level* attribute of the sequence point is an integral value used in the Iterative Topological Ordering algorithm to ensure valid order of the operations.

10.1.1.4 Conjugated Sequence Points

We say the two sequence points α and β are conjugated, if **psp(**β**)** = α and **nsp(**α**)** = β or **nsp(**β**)** = α and **psp(**α**)** = β. We say that α and β are head and tail of an conjugate sequence-point chain if there exists a possibly degenerate chain of sequence points $\gamma_1 \ldots \gamma_n$, $\gamma_1 = \alpha$, $\gamma_n = \beta$ such that $\gamma_i$is conjugated with γ_{i+1} for all i for which γ_{i+1}exists and neither **psp(**α**)** is conjugated with α nor **psp(**β**)** is conjugated with β. We say the the γ_i are members of a conjugate sequence-point chain.

All members of the same conjugate sequence-point chain have the same level. The level of such chain is determined as the maximum of the levels of the chain head's **psp** and the chain tail's **nsp**[4] plus one.

[1]Either we can assume it because there is a sequence point at the end of the preceding statement or there is a sequence point at the beginning of the function containing the affected statement.

[2]This is the case of the built-in comma operator. This case of user defined comma operator is reduced to the case of a function call discussed elsewhere.

[3]Here too, we consider only builtin operators.

We can evaluate the left subexpression first yielding the sequence point ordering $\alpha\gamma\delta\beta$. Alternatively, we can evaluate the right subexpression first, yielding the ordering $\alpha\delta\gamma\beta$

[4]If either does not exist it is assumed to have level minus one.

10.2 SA Context

A context for the SA process needs to be stored and managed, because during the parsing a contextual information needs to be saved, that cannot be saved by the means of bison symbols semantic values. The context thus maintained is divided into three disproportionally large parts. These parts are discussed below.

10.2.1 AS Context Part

The context maintained in the AS part consists of an accessibility specifier, and a current scope expressed as an instance of `as_name` type. The scope is used in forming a pseudoqualification for unqualified names and is referred to by a helper macro `CURR_CTX` in the bison generated parser.

10.2.2 SS Context Part

The SS context part contains SS structures for concepts parallel to those contained in AS control context parts. The access specifier is represented by an enumeration declared in class `ss_access_specifier`. The scope is represented by an instance of `ss_decl_seq` class.

10.2.3 SA Context Part

The SA part contains contextual information that is needed for the SA process itself. Via the use of this part of context a passing of data from left to right in the bison parser is possible, albeit the bison parser uses $LALR(1)$ analysis thus allowing only fully synthesised attribute grammars. This part of context can be created empty and filled afterwards.

This part contains a last processed declaration specifier sequence and some helper context for the statement handler. This part should also contain a stack of currently opened breakable statements (i.e. switch and iteration statements) that are used to determine the target into which the control is transferred when a `break` or `continue` statement is encountered. This stack, however, (alongside with the break and continue handling) is not implemented yet.

10.3 Managing Contexts

The context described in 10.2 is maintained by `sa_context_manager` class which is a singleton holding a stack of `sa_context`s. It's methods `push()`, `pop()` and `current()` provide a convenient interface for manipulating the contexts in the program.

10.4 Statements

10.4.1 Overview

The goal of these algorithms is to create SS structures for statements with correctly connected sequence points, appropriate labels and all other information needed to generate pseudoinstructions. It would be easy to implement this as a direct transformation from AS structures which are created clearly in parser actions. The problem is that C++ requires many SS structures to be ready to use earlier than the AS structures are created. For example, the AS representation of a compound statement is created when the parser reduces by the rule for a compound statement - on the enclosing brace. However, if a declaration appears in the compound, there must be a scope on the SS level where to add it (otherwise the identifier lookup would not work). Therefore the SS structures for scope have to be created on the *opening* brace of that compound. Similar situation occurs with many other statements. Correct creation of SS statement structures therefore requires touches to grammar and adding special actions.

All algorithms described in this section are implemented in `sa_statements` class and some small helper classes. The code can be found in `lestes/lang/cplus/sem` directory in `sa_statements*` files. The grammar is in
`lestes/lang/cplus/syn/parser.y`. Small touches have been done to other files, such as some LVD files.

10.4.2 Parser actions

It is easy to add an action to be executed when a statement is completely parsed. We find the appropriate rule in the grammar and simply add a call to our function. Assume the following rule.

```
compound_stmt: TOK_LEFT_BRACE stmt_seq_opt TOK_RIGHT_BRACE
    { $$ = as_compound_statement::create(loc($1), UNWRAP($2)); };
```

The action in braces is performed when the token TOK_RIGHT_BRACE is read, so we add a call to our `leave_scope()` there. However, we need to perform another action on the TOK_LEFT_BRACE token. So we add a new nonterminal which derives to the TOK_LEFT_BRACE and replaces the token in the previous rule.

```
compound_stmt:
  _enter_compound_action stmt_seq_opt TOK_RIGHT_BRACE
    { $$ = as_compound_statement::create(loc($1), UNWRAP($2));
      sa_statements::instance()->leave_scope();
    };
_enter_compound_action: TOK_LEFT_BRACE
    { $$ = $1;
      sa_statements::instance()->enter_scope(loc($1));
    };
```

The `$` signs indicate usage of attribute grammar technique (see bison documentation) to get information from parsing the symbols used in the rule.

10.4.3 Sequence points and other common contents

All SS statements have the following fields and these are filled similar way.

`location` Location of the statement in the source code. Typically it is a location of the first token of the statement and it is passed from parser.

`labels` List of labels pointing at this statement. It is created empty when creating the SS statement and some other actions (handling labeled statements) may push labels into it.

`parent` Every statement is placed in a scope. This field links to the nearest enclosing scope.

`psp` Previous sequence point.

`nsp` Next sequence point.

`sequence_points` List of sequence points within this statement.

The last three fields have to be handled carefully otherwise the backend algorithms get confused. This is described in details for each statement separately below. Two new sequence points psp and nsp are always created upon creation of a statement and they are inserted into its list of sequence points. They are correctly connected to each other and to current chain. See the `ALPHABETASPS` macro in `sa_statements.cc` for details.

10.4.4 Expression statement

The action called from parser upon expression statement takes an AS expression as its parameter. The expression has to be transformed to an SS expression and encapsulated by an SS statement. First of all, the two sequence points are prepared for the SS statement. The expression transformation is done by the `sa_expression_converter` visitor. If the expression is empty, this visitor returns `NULL` and the statement is simply discarded, since it has no effect. Otherwise the `sa_deconstruct_spse` interface is called to perform the transformation. The sequence points passed to it are the PSP and NSP created earlier. Then the instance of `ss_expr_stmt` class is created and pushed to the end of current compound statement.

10.4.5 Return statement

Handling the return statement is quite similar to the expression statement except that the expression is converted to the return type of current function. The `ss_type` is extracted from the declaration of current function and passed to the `sa_expression_converter` visitor as a parameter of its alternative `create()` method. If the conversion sequence doesn't exist, the visitor reports an error to user.

10.4.6 Compound statement

On the opening brace of the compound statement, a new scope is created and the SA context10.2 is switched into it. On the closing brace, the current context is simply removed from the context stack.

Opening a new scope requires:

- Creating a `ss_decl_seq` + `ss_compound_stmt` pair.
- Filling in a (dummy) declaration for that `ss_decl_seq`.
- Pushing the compound statement into current compound.
- Connecting the sequence points into the current chain.
- Creating and filling in a new SA context.
- Switching to the new context by pushing it on the stack in the context manager.

At the beginning of the `enter_scope()` method, there is a trick with the `ss_compound_stmt::behavior` flag. It is useful for statements containing their own sub-scopes such as bodies of `while`, `for`, and `if`. It is discussed below by the `if` statement.

The compound statement has an additional sequence point used by the ss2pi part for destruction of objects that were created within the compound. Upon creation of the compound statement, this sequence point is connected between the PSP and NSP. Each new statement that will be inserted into the compound statement has its sequence points connected to the chain between the destructor sequence point and its nearest predecessor.

10.4.7 Function body

To enter a function body, the new scope has to be created similarly as for the compound statement. A few additional actions are performed. First, the *parent* of the new scope is not the actual scope but the scope with function parameters. It can be obtained from the function declaration which is the last declaration remembered (see declaration statement). After creating the scope, the declaration of current function is stored into `current_function` variable for future use (e.g. for the `return` statement). The last action is adding a *fake declaration* for each parameter into the newly created `ss_decl_seq`. This is the way to avoid redeclaration of parameters in the function body.

10.4.8 Declaration statement

We have a *broadcasting* mechanism to let the statements generating code know about each new type declaration. Class `decl_stmt_listener` is attached to this broadcast and its `run()` method is called on each new SS declaration. Each such declaration is recorded into the `last_declaration` variable for future use. If it does not declare an object but just the type, an SS declaration statement is immediately created and inserted into current scope. Otherwise we wait for an initializer that could follow. Assume the following line in the source code.

```
int i, p*,j=3, k(7);
```

There will be four broadcast events with type declarations: `int`, `int*`, `int`, `int`. After each, the parser will reduce by the rule for `init_dcltor` nonterminal. That is the point to catch a declarator with initializer (or without). There are three types of initialization that are distinguished in the parser.

- Declarator without initializer. (`int i;`)
- Declarator initialized by a copy constructor. (`int j=3;`)
- Declarator initialized directly by calling a constructor. (`int k(7);`)

For builtin types there are no constructors, but the syntax is the same and the SS structures are filled the same way. If there is an initialization, we have to convert each expression to its SS representation (as described by the expression statement). In case of copy initialization, a conversion to the target type is requested. Each expression from the initializer is transformed and filled into the `args` field of the resulting declaration statement.

10.4.9 If statement

According to the C++ specification, the condition in `if` statement may contain an object declaration. The declared object is visible in both the "then" and "else" substatements and its name cannot be redeclared there **[6.4/1-3]**. We solve this by encapsulating the `if` statement with a special scope and the declaration (if present) is inserted into it as a normal declaration statement. The "then" and "else" substatements are always compound statements. To avoid redeclaration of the object from condition, we insert a *fake declaration* into each of the subscopes (only into the `decl_seq`, no statement is created).

The statement

```
if (int i=1) stmt1; else stmt2;
```

is transformed to

```
{ int i=1; if(i) { int i; stmt1; } else { int i; stmt2; } }
```

Note that the declarations in substatements are of the type `ss_fake_declaration` and there are no declaration statements for them.

The encapsulating scope is entered immediately on the TOK_IF token. Then after parsing the condition, the `if_head()` function is invoked to insert the declaration (if present) and create the SS structure for `if` statement. We have to create the `if` statement before we enter any of the substatements, because of the properties of the grammar.[5] Note that the substatements are created before the parser reaches them. Therefore when it reaches one of them, no new scope is created but the context is switched into the prepared scope. This is done by the `enter_subcompound()` routine. It also inserts the fake declarations if necessary.

The specification says that `if (cond) stmt;` is exactly equivalent to `if (cond) { stmt; }`. The substatement is always a compound statement on the SS level. So in the first case we have to create an encapsulating compound around the substatement. However, if we wold do the same in the second case, the trick with a fake declaration would not work. So if there is a compound statement given by the user, it has to be ignored. That is the reason for the trick with `ss_compound_stmt::behavior` flag. It denotes whether the new compound has to be created.

10.4.10 While statement

Handling the `while` statement is similar and easier than handling `if`. There can also be a declaration in the condition, so we create an encapsulating scope and use the same trick with fake declaration.

```
while (int i=j--) stmt;
{ int i=j--; while(i) { int i; stmt; } }
```

The object declared in the condition is constructed and destroyed in each iteration of the loop **[6.5.1/2]**. Therefore the ss2pi transformation has to perform a jump to the outer left brace in each iteration, not only to begin of the `while` statement.

10.4.11 For statement

The `for` statement is a bit more complicated than the `while`. There can be up to two declarations in its head part. Therefore it is always encapsulated by two special compound statements and several fake declarations are inserted into nested scopes.

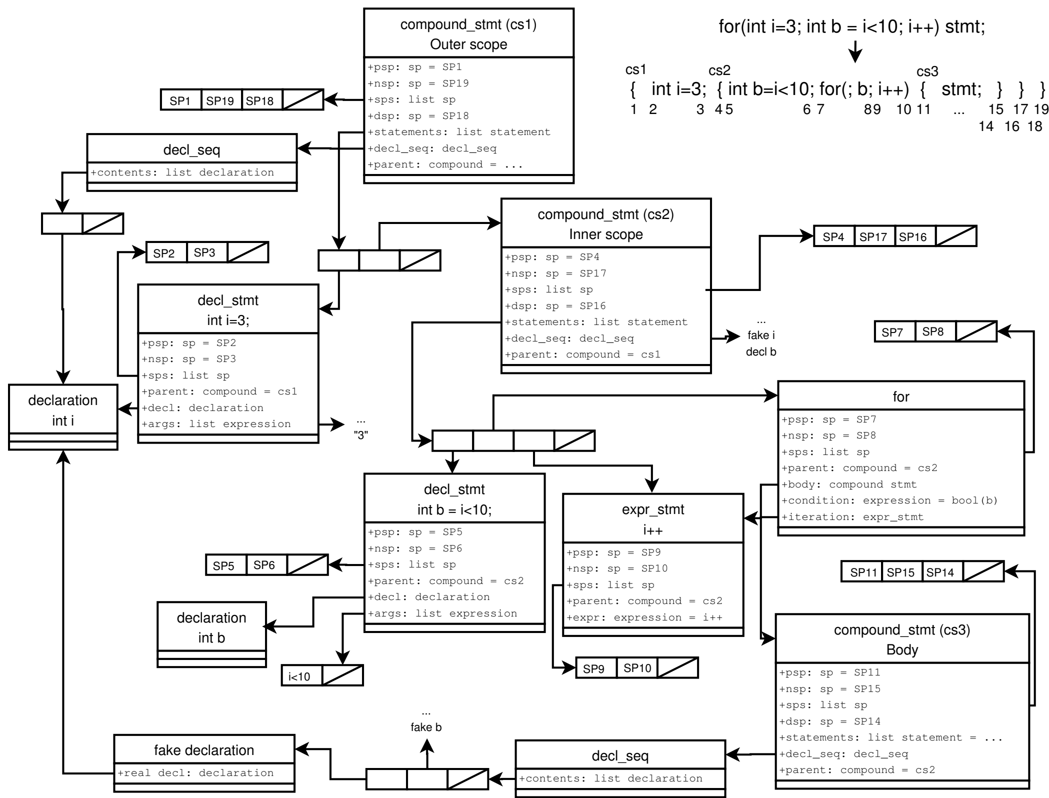
```
for(int i=0; bool b=i<10; i++) stmt;
{int i=0;{int i;bool b=i<10;for(;b;i++){int i;bool b;stmt;}}}
```

The initialization statement is placed completely to the outer scope and not mentioned in the SS class for `for` statement. The iteration expression (if present) is transformed to an expression statement and connected by its sequence points after the loop body just before the end of the inner encapsulating compound.

The condition can be omitted. **[6.5.3/2]** Then the `true` condition is created and inserted into the resulting `ss_for` class.

The following figure is a simplified illustration, how the SS structure for the `ss_for` statement looks like after the SA analysis.

[5] The parser solves the dangling-else problem and therefore the grammar rules are not always straightforward.



10.4.12 Do statement

The `do` statement can be handled much easily than the other two loops, since it cannot contain a declaration in it's condition. **[6.5/1]** There is no need to encapsulate the `ss_do` with a special compound. On the `TOK_DO` the new for the body compound is created along with the `ss_do` class. On the end of parsing the `do` statement, the condition is filled. The expression is transformed to SS and converted to `bool`.

10.4.13 Switch statement

The `switch` statement has the same type of condition as `while` has except that it is not converted to `bool`. Therefore the condition may contain an object declaration and therefore the `switch` has to be encapsulated by special compound statement. The body of a `switch` is a normal statement except that it may contain `case:` and `default:` labels (see labeled statement below).

According to **[6.4.2/4],** the `switch` statements can be nested and the inner labels are associated with the nearest enclosing `switch`. Therefore we need a stack of `switch` statements in the SA context 10.2. On the `TOK_SWITCH` token we push the new statement to the stack and we pop it on leaving the `switch` body statement.

10.4.14 Continue statement

The `continue` statement has the only special field - link to an iteration statement that is to be continued. It is the nearest enclosing iteration statement. Therefore the iteration statements (`do`, `for`, `while`) are stored in a special stack in the SA context 10.2. Then it is easy to determine the one that belongs to this `continue` statement.

10.4.15 Break statement

Handling the `break` statement is similar to `continue` except that a different stack is used. Breakable statements are not only the iteration statements but also the `switch`, so we need a different stack in SA context 10.2.

10.4.16 Try and catch statement

The exception handling is not implemented in this version of Lestes compiler.

10.4.17 Labeled statement

Every label belongs to a single statement that follows just after it. There are three kinds of labeled statements.

- *Identifier labeled statement.* In the function declaration there is a map of identifier labels along with the corresponding statements. When we see a new labeled statement, we have to add the pair into this map. If the label is already there, report an error. **[6.1/1]** The new `ss_label` is also added to the list of labels attached to the statement.
- *Case labeled statement.* The `case` label has an expression to be converted to SS. The `ss_case` class contains also a link to the `switch` statement that this label belongs to. It is determined from the stack in context similarly as with the `break` statement.
- *Default labeled statement.* This is not different from the `case` labeled statement except there is no expression to be converted.

10.4.18 Goto statement

The `goto` statement contains an identifier of a label. The label may lay before the `goto` or after it. In this phase we do not try to find it. We simply create a new label with that name and store it in the `ss_goto` class. The subsequent parts of the compiler have to go through the list of identifier labels in the function and find it by the name.

10.4.19 Work to be done

The following statements are to be implemented in the future.

- Labeled statements and `goto`.
- The `do` loop statement.
- The `switch` statement.
- The statement stacks to correctly handle the `continue` and `break` statements.
- Exception handling, the `try` and `catch` statements. (`throw` is an expression, not statement.)

Except for the exception handling, the possible implementation is suggested above and should be clear. The `for` loop statement is already correctly passed to the SS2PI part, but not yet handled by it.

10.5 Declarations

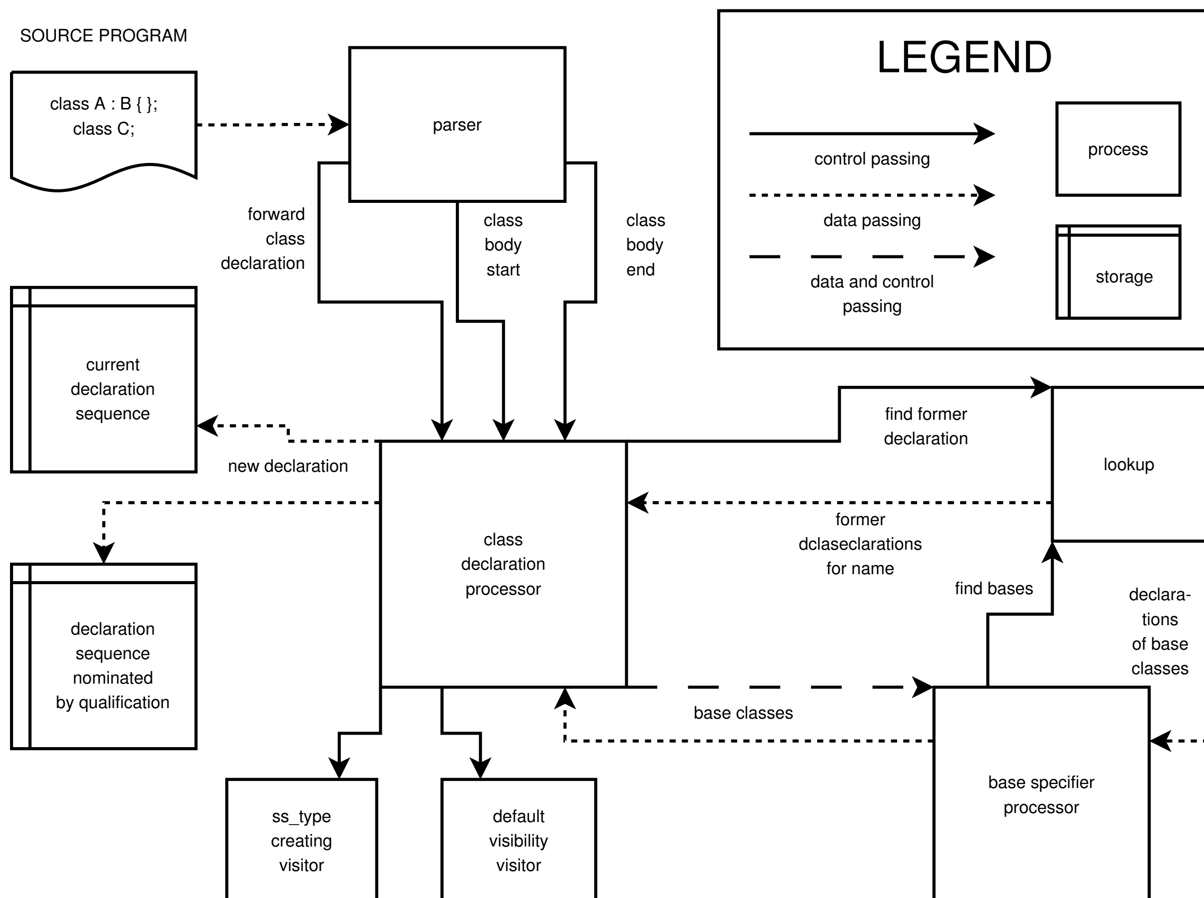
The handling of declaration is a very delicate process, as the declarations need to be introduced into their appropriate scopes before their complete declaration is found. The handling is further complicated by the need to find previous declarations for the same name in order to discern the possibility to declare the new meaning, or to unify the new information with the old information.

10.5.1 Class Declarations

The class declarations are the declarations that exhibit the most peculiarities that need to be handled. Firstly, the class contents can be accessed via qualified names, a feature peered only by the namespace declarations. Secondly, the classes can be invisible, thus requiring separate lookup in order to determine whether a class declaration shall be coupled with a pre-existent declaration. Thirdly, the contents of the class is subject to accessibility rules, a feature unpeered by any other declaration. Lastly, the base classes impose yet another difficulties to the lookup implementation.

Class declarations do not come to the transformational process as declarations of its own right. They do come as a member of some declaration specifier sequence occurring in a simple declaration. Additionaly they have to be handled early, much earlier than the entire simple declaration in which they occur.

The SA transformation of class declarations begins as soon as the class head is read. The first goal of the transformation is to establish a new scope for the class. Secondly, a declaration entry for the class is inserted into the current scope. Only after these steps are completed, the parsing of the class interior can begin.



10.5.1.1 Class Visibility and Invisibility

The classes come in two flavors. The ordinary flavor is the visible flavor. These declarations are found during the qualified and unqualified lookup **[3.4].** The visible class declarations can be introduced in two different forms: the so called forward declaration in the form `class name;` and the full declaration, that contains the body of the class, enclosed in braces. The invisible case is the more interesting one. These declarations come from locations that are not directly in the scope the class is declared into. The first source of the invisible classes is the friend declaration in another class. The friend declaration introduces a possibly not visible name into the innermost enclosing namespace scope. The second source of the invisible declarations is the use of the elaborate type specifier in a context different from the context of the forward declaration.

10.5.1.1.1 Example

```
class X {
    friend class Y; // 1
    struct D * data; // 2
    int foo(struct Z *); // 3
};
int bar(struct Y *); // 1 (+)
class D { // 2 (+)
    struct Z * z; // 3 (+)
};
Y * y; // 1 // error: Y is not visible
```

The most notable thing is that in the lines marked by the (+) marker the class needs to be looked up, even if it is invisible. This is required so that the types can be correctly matched.

10.5.1.2 Accessibility of Class Members

The accessibility of the members is directly expressible by the use of a flag determining the accessibility status, public, protected, or private. This change, however, has an impact on other declarations, because the flag needs to

be associated with the member declarations.

This requires creation of another base class containing a field to hold this information. This class is called `ss_befriendable_declaration`.

During the SA of classes, the access specifier has to be determined by means of transforming the keyword that starts the class declaration into the appropriate SS representation of the access specifier. This constitutes so-called default accessibility and the mapping is as follows: if the keyword is `struct` or `union`, the default accessibility is public whereas if the keyword is class the default accessibility is private. The default accessibility applies also to the accessibility of base classes provided that the accessibility of base classes is not marked explicitly by keyword `private`, `public` or `protected`.

The default accessibility is also recorded in the `sa_context_managers`'s context for the class being processed as it's needed for creation of class members before the first access declaration is encountered in the body of a class. The access declaration replaces the recorded information whenever it's encountered.

The contextual representation of access specifier is stored in both AS form and SS form in order to provide rapid access to both forms during the process of parsing and semantic transformations.

10.5.1.3 Base Classes

The base class specification is processed in the order given in the source program. Each of the base classes mentioned in the base clause specifier is looked up and checked that it's not the union, for unions shall not be base classes of any class or union. It is an error when the base class is not visible, accessible, or found. The currently handled class is inserted to the list of descendants of all the base classes. This is required in order to support member pointer conversions elsewhere in the SA process 10.7.4.

Each base class is associated with an access specifier either coming from the default one or directly specified and the information whether the given base class is a virtual base class or not.

10.5.1.4 Class Completion

At the end of class body, the SS type representing the class is modified to show that the class it represents is a complete type. The context is switched back from the context of the class to the context that was actual before the class body was begun to be processed.

10.5.1.5 Declaration of Class Members

The transformation of class member declarations proceeds along the general pattern of the namespace-level declarations. The situation is nevertheless different in certain aspects.

Firstly, the set of allowable declaration specifiers is changed. Secondly, there are certain novel kind of declarations not occurring elsewhere.

The storage class specifier *mutable* can be only applied to non-static data members of a class.

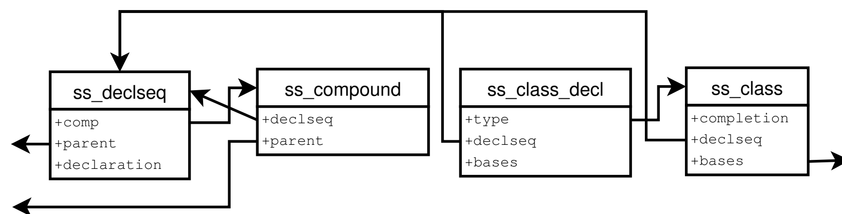
10.5.1.5.1 Friend Declarations Friend declarations look like the most common kind of declarations occurring on a class scope. They differ in the presence of *friend* declaration specifier. They also differ in the way they are handled. While non-friend declarations introduce names into the containing *class-scope,* friend declarations introduce the declared names to the innermost enclosing namespace scope. Yet another difference can be observed. Those names are declared in a manner that makes them invisible to the ordinary lookup. On the other hand they can remain visible to the argument dependent name lookup.

10.5.1.5.2 Old-style Visibility Promotion This particular kind of declarations occurs on the class scope exclusively. They consist of an unadorned qualified name. Their sole purpose is to adjust visibility of the declared name. For example a name declared protected in a base class can be made public. Also the names declared in some base class inherited via private or protected inheritance can have its access restrictions lifted by means of this kind of declaration. The effect of this type of declarations can be achieved by means of a using declaration. This type of declarations is deprecated because of its idiosyncrasies. Both old-style visibility promotion declarations and using declarations on a class scope are handled equivalently. They do share one section describing their implementation.

10.5.1.5.3 Using Declarations The using declarations are a new alternative to the previous kind of declarations. They have a broader scope of application mainly in template declarations, because they allow for use of the keywords *typename* and *template* in them. For each such using declaration an alias declaration is formed and added to the containing scope. See ***10.5.3.***

10.5.1.6 Implementation

For each class head a new *declaration_sequence* is created along with the associated *ss_compound_stmt*. An instance of *ss_class_declaration* is created along with a new type object representing the class itself. The type of the class is not generally complete by the time the end of a class head is encountered, therefore the completion time of the type is set to a value denoting that the class type is not complete. The resulting schema resembles the following:



Also the default visibility has to be determined and inserted to the parser's context.

10.5.1.6.1 Implementation of Class Member Declarations The class member declarations are handled by a process similar to the process handling simple declarations on the namespace scope

10.5.1.6.2 Friend Declarations These declarations are inserted into the innermost enclosing namespace as if they were declared there. The difference is that they have their visible_since property set to infinity. In addition to that the list of friendly classes is updated on the target (i.e. the friend's) declaration.

10.5.1.6.3 Old style Visibility Promotion This is handled in the same manner as the using declarations.

10.5.1.6.4 Using Declarations Using declarations on a class scope are handled the same way as their namespace-level relatives. In the process of ordinary name lookup they are ignored.[6]

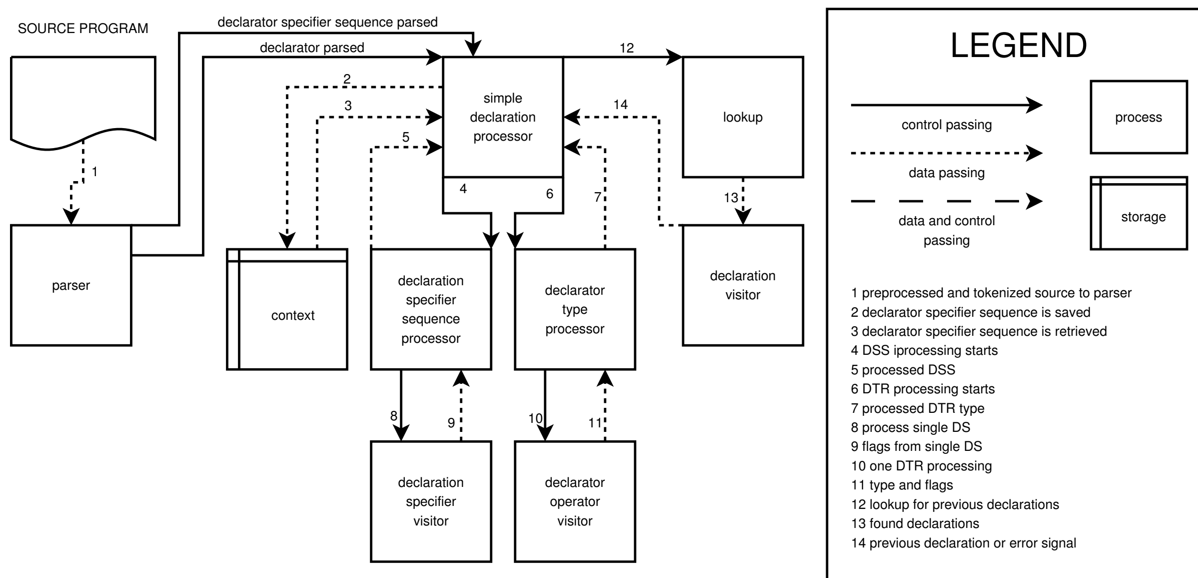
```
class A { public: int i; };
class B { public: int i; };
class C: public A, public B {
public:
        using A::i;
};
C *p;
```

Given the declarations from the preceding example, the expression *p->i* is ill formed, because the lookup does not find an unique declaration of *i*. The using declaration only has an effect on access checking.

10.5.2 Simple Declarations

A simple declaration syntactically consists of a declaration specifier sequence and a declarator. The declaration specifier sequence needs to be analyzed first. In the syntax of the simple declaration one declaration specifier sequence can be shared by multiple declarators (for example `int i, j`). The declaration specifier sequence is analyzed only once for all the subsequent declarators.

[6] I.e. they cannot be used to resolve ambiguities concerning member name lookup.



10.5.2.1 Overview of the Analysis Steps

The kind of the declared name is to be determined first, then according to the kind the declaration is created and inserted to it's appropriate scope. The kind is determined by analysing the declaration specifiers and the declarator. Certain declaration specifiers are sufficient to determine the kind completely whereas in other situations the declarator must also be taken into account (e.g. if the declaration specifier `typedef` is encountered in declaration specifier sequence then the kind of the newly introduced declaration is `typedef` declaration. In the absence of any storage class specifier ans `typdef` specifier, the declarator has to be examined in order to find out whether the declaration will be an object declaration or function declaration).

The second step in processing the simple declaration is to find out the type of the name being declared. This is performed by first determining the basic type from the declaration specifiers and then augmenting the basic type as dictated by the declarator operators present in the declarator.

Only after the type has been successfully determined it is possible to lookup any previous declaration of the name as to discover whether the current declaration conflicts with a previous declaration or is an extension thereof. [7]

After these steps are completed, a new declaration is introduced to the current scope or the scope determined by the qualification of the name being declared if necessary.

10.5.2.2 Handling of a Declaration Specifier Sequence

Each of the declaration specifiers is checked whether it can be combined with other declaration specifier present and whether it's allowed in the context where the declaration occurs. E.g. the declaration specifier `static` cannot be combined with the declaration specifier `typedef`. By analyzing the declaration specifiers it is possible to determine a type that can be subsequently used as a basis for the type of the name definition declared by the following declarator.

In the course of processing the declaration specifiers errors are reported whenever a declaration specifier that cannot possibly occur on the given scope is encountered (e.g. the declaration specifier `explicit` cannot be used outside of class bodies).

10.5.2.3 Handling of Declarators

The declaration specifier operators are processed in order to determine the complete type of the name being declared (e.g. assuming that the basic type coming from the declaration specifier sequence process is `int` then given the declarator `(*f)()` the resulting type is a pointer-to-function taking no arguments returning int. The declarator operators are processed in reverse order because the last declarator declaration operator has to be applied to the basic type first and every declarator operator shall be applied to the result of the following declarator operator application.

[7] This part of simple declaration handling is not completely implemented yet.

10.5.2.4 Looking Up the Previous Declarations

Having determined the kind and type of the name being declared, it is possible to perform lookup in the scope that is about to receive the new declaration in case it would be created. The lookup shall resolve the using declarations but it shall ignore the `using` directives in order to confine the lookup into that single scope. If the declaration kind is `typedef` declaration, all non-type declarations are ignored during the lookup, whereas in the remaining cases the type declarations are ignored. For a program to be well-formed, it is required that the result of the lookup contains only those declarations that are compatible in kind with the declaration which introduction is considered. (That means that in the case of function declarations it's an error if the lookup finds an object declaration and vice versa). [8]

In the case of function declarations, the set of found declarations can have multiple elements. These are examined in order to find a function with a prototype matching the prototype of the function from the simple declaration being processed. If a declaration is found, but it's returned type does not match, the program is ill-formed, because it's not permissible to overload two functions in a way that would make them differ only in a return types. If exactly one declaration is found, the new information coming from this simple declaration is combined with the information stored in the found declaration, as discussed below. If no matching declaration is found a new declaration is created.

In the case of object declarations, the declaration found shall be declaration that was declared using the `extern` storage class specifier with no initializer unless the current declaration contains the `extern` specifier. In any case the types of the declaration about to be introduced and the found one, shall be the same. If a previous declaration was found, it is modified in order to contain the combined information itself and a new declaration.

In the case of `typedef` declaration, the previously declared declaration shall be also a `typedef` declaration to the same type or should be a class declaration. If it is a class declaration, the type of the class shall be the same as the type of the new `typedef` declaration to be introduced. [9]

10.5.2.5 Introducing and Merging Declarations

If the currently processed declaration is a declaration of a function and a previous declaration was found in the preceding step, a new scope is created for the parameters, the parameters are added to it, while simultaneously checking that the default values supplied in the parameter declaration do not conflict with the previously available set of default arguments, and that do not introduce gaps in the sequence of default arguments. The previous scope for the parameters available as the value of the field named `parameters` in the `ss_function_declaration` is recorded in the list called `historical_parameters`. The new parameter scope replaces the previous parameter scope as the value of the `parameters` field.

If the currently processed declaration is a declaration of a function and no previous declaration was found, a new declaration is introduced by filling all the necessary fields, most notably a new scope for parameters is created and filled with the parameters specified in the param-decl-clause of the currently processed declaration.

In the remaining cases, the processing is quite straightforward and was discussed above except for minor implementation issues.

10.5.3 Usings and Aliases

It is necessary, in certain situations, to create another entry for an already existing declaration. The reasons for such declaration aliasing fall into two categories. The first category of aliases consists of aliases created because of using declaration. The second category of aliases contains aliases of declarations that should no be redeclared in a particular declaration.

The second category is, strictly speaking, unnecessary. The redeclaration avoidance could be conceivably using other sophisticated rules. An example of such alternative approach would be to associate a non-negative number with each scope. The number would tell the number of enclosing scopes that should be considered while discerning whether a particular declaration constitutes a forbidden redeclaration or not. This alternative approach, viable as it might be, imposes additional burden to the lookup, because it necessitates limited depth lookup in addition to the fully recursive lookup and the lookup confined to a given scope.

[8] The lookup result requirement checking described here is not performed yet.

[9] The matching of the previous declaration in the typedef case is done only for the non-class case. Additionally, the error is incorrectly not reported in the current implementation.

10.5.3.1 Using Declarations

Using declarations constitute declarations for the name into the enclosing scope. For the purposes of redeclaration it is an error to declare a non-redeclarable name via an using declaration. Additionally it is an error to redeclare a non-redeclarable name even if it was declared by an using declaration in the first place. As the using declaration introduces a set of names into the current scope, the using declaration could be ill-formed even though there exists a declaration of the target name that could be redeclared.

```
namespace X {
        class A { int i; };
        int A(int);
        int A(double);
        int f(int);
        int f(double);
}
namespace Y {
        typedef int A, f;
        using ::X::f; // ok
        using ::X::A; // ill formed. ::Y::A typedef clashes with class ::X::A.
}
```

10.5.3.2 Implementation

In a using declaration in the form **using** *qname***;** a new declaration, represented by *ss_using_declaration* class instance, is added to the scope containing the using declaration for every member of the resulting set of declarations obtained by the qualified lookup of the qualified name *qname*. Validity of such declaration (nonviolation of the "cannot be redeclared" rules) is checked for every such new *ss_using* declaration [10]

All the coming declaration information is copied into the *using_declaration* except for some technicalities concerning the declaration instance, not the declaration meaning. Pointer to the "real" declaration is also filled appropriately.

10.5.3.3 Protective Aliases

This second kind of alias declarations is used to prevent redeclaration of parameters and variables declared in the head of *if*, *while*, and *for* statements. The creation of this type of aliases is different from that of **using**. Firstly, there is no need to check whether the declarations exist, for they are introduced into an empty scope. Secondly their form is different, for they do contain only one link to the "real" declaration, because the case "using of a using" does not exist. Certainly an example can be crafted, that would seemingly require the "no redeclaration check" to be performed on these aliases. Example:

```
int foo()
{
       int x;
       for(int i=3; int i=x<2; x++)
             /* empty */
}
```

Here the second declaration of *i* clearly violates the no-redeclaration rules. The flow of actions rightly detects the violation. First, a new scope is created to hold the declarations in the *for-init-statement*. Second, the first declaration of *i* is processed and introduced to the aforementioned scope. Third, yet another scope is created for the condition declarations <u>and</u> all declarations of the first scope are aliased into it. Fourth, the second declaration of *i* is processed finding clash with the alias produced before. Only after these steps the aliasing of the second declaration would be performed.

[10] Or should be in the future implementation.

10.6 Deconstruct SPSE

10.6.1 Overview

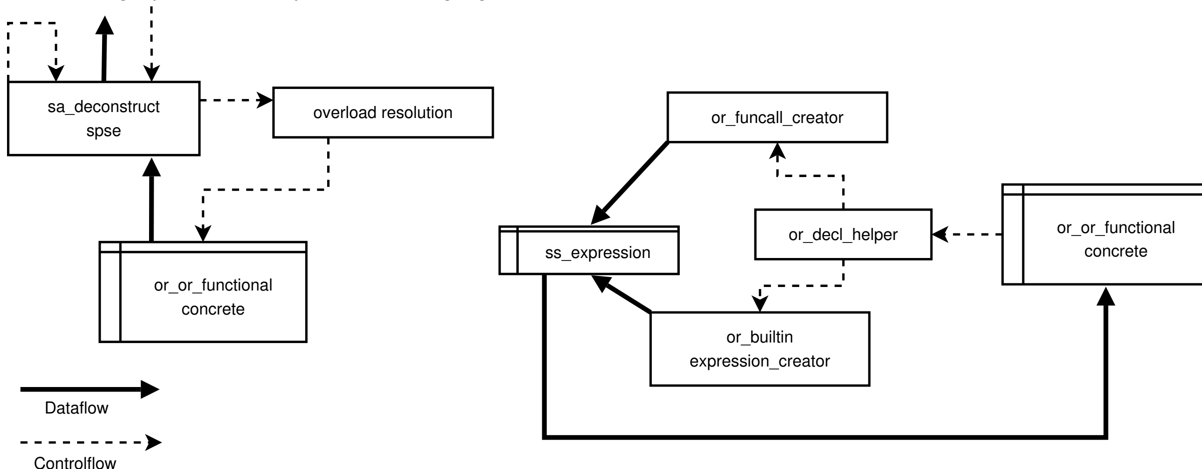
The expression transformation part of the SA is called Deconstruct SPSE. Although the name for the process predates the global SA structuring, it well suits the process of disassembling (deconstructing) AS structures and reassembling them into their SS counterparts, of which the most notable to the further states are sequence points (SE) and side-effects (SE).

When describing the Deconstruct SPSE process, the term *"Overload resolution"* is used. The details about Overload resolution implementation are described later, here we describe only the basic concept, needed to understand the following text.

The Overload resolution process is triggered whenever there is a need to chose a function that has to be called in the given context (this also includes deciding whether the builtin operator (if such exists) or the user-defined function should be called). The overload resolution consists of two parts: candidate finding, i.e. the selection of possible meanings for an operator or possible functions in the case of function call, and best such meaning selection, i.e. finding the meaning or function that necessitates the least effort to perform. The rules for such selection are elaborate and they are discussed in 10.7 in more detail.

10.6.2 Implementation of expression transformation

The process of transforming the expressions from Abstract Syntax structures to the Semantic Structures intercode can be roughly described by the following figure



First, there is a `sa_deconstruct_spse` visitor, which is visitor defined on `as_expression` type, which is a structure, created by syntax analysis and representing expression in abstract syntax layer. The main task of this visitor is to perform an overload resolution on the name of the given operator (we can get the name easily, as the visitor mechanism tells us which successor of `as_expression` is currently being processed, so corresponding class of type `ss_operator_<something>`, which is a class representing operator name for every possible operator, can be created and passed as argument to overload resolution.

Overload resolution process creates functional `or_or_functional` (for the explanation of mechanism of functionals, see overload resolution chapter), representing expression for chosen function (or special functional `or_or_functional_ambiguous`, representing the ambiguous conversion; this functional should also return list of viable functions, to be able to show to the user candidate functions which were taken into account when trying to find best viable function) and returns it back to `sa_deconstruct_spse` visitor. This visitor then returns this functional, which is later used by the caller of the whole expression transformation process, at the time when the creation of actual `ss_expression` is needed. There is rationale for doing it in this non-straightforward way, with postponed `ss_expression` creation. That is that during the time when overload resolution is running and expressions are transformed from AS to SS, there might not yet be complete information about sequence points, which the expression should have. This information can be known as soon as the whole expression is converted, but not sooner. So the deconstruct process works in a way that it creates a hierarchical structure of *creators* of `ss_expressions`, which copies the hierarchical structure of the expressions, but the actual `ss_expressions` are created by these creators later, when caller can pass sequence points to the whole process.

This process is differentiated a bit according to the type of as_expression which is being processed. There are groups of operators that can be handled in the same way, which is done using templated methods in visitor class

sa_deconstruct_spse.

- The templated method construct_bin_op(), is used to create functionals for the majority of "usual" binary operators, including for example assignment, arithmetic operators (+,-,*,/,&,+=,-=,*=, etc). In this phase it is not necessary to distinguish if new sequence point and/or sideeffect needs to be created. Handling of this can be safely postponed until the actual creation of ss_expression, as this is the first structure that is requiring the sequence point/sideeffect information. Therefore handling this group of binary operators involves just recursively handling left and right subexpressions and then performing overload resolution (to determine "what to do"), using results of left and right deconstruct, obtained from recursion, as parameters for the overload resolution.

- There are two different templates, construct_unary_op_create() and construct_unary_op_nocreate. To understand the difference between these two templated methods, we will have to understand how C++ Standard distinguishes, for purposes of overload resolution, between postfix and prefix increment/decrement operators. This is described in **[13.5.7]** and also a few remarks in **[13.6]**. Overload resolution must have possibility to distinguish between postfix ++ and prefix – operators, to be able to choose the correct function from candidates (which can be builtin and user-defined functions, mixed together). The Standard defines, that for the purposes of overload resolution, the postfix operator++ (or operator-- respectively) has a prototype

  ```
  T operator++(T&, int)
  ```

  in contrary to the prefix one, which has prototype defined by Standard as

  ```
  T& operator--(T&)
  ```

  From this, we can not only see the fact that the postfix case has two-parameter prototype, but also that the postfix operator does not return lvalue, but the prefix one does. This is why there is need for two separate templates - the "nocreate" one just calls the overload resolution as usual, however the "create" one has to create the second ("right") parameter manually. It creates signed int literal with the value of 1 and uses this as second parameter when calling overload resolution.

- The next method to be discussed is visit_as_expression_comma(). It differs from previous ones in the sense of handling the result of overload resolution. As comma is defined for every pair of types of expressions, if there is no function chosen by overload resolution, this doesn't mean that error should be reported to user. The special functional, representing the comma expression, is created instead. It is necessary to note, that there are many other operators, which are, by **[13.6]** defined "for all types" or "for all object types", etc. This is not handled in the current version of the compiler, for most of the operators only builtin types are considered. Handling of them could in future be added by in the similar way to how comma operator is currently handled.

- The next notable method is visit_as_expression_unary_amp(). For the purpose of passing the operator/function name to the overload resolution, the ss_operator_band (which comes from bitwise and) is created here, even though the unary & operator has nothing to do with bitwise and. But for the purposes of overload resolution this is enough, as bitwise and can be easily distinguished from unary & by the number of arguments, so the overload resolution will always return the operator with correct semantics.
  In the case of overload resolution not finding any suitable candidate function, the special functional for the address_of expression must be created manually (special functional is required, as there is no declaration for anything like "address_of" function, and the creator must deal with it), as the unary & is also the case of operator accepting infinite number of operands (see **[13.3.1.2/3]**). The type passed to this functional must be "de-referenced" and "de-pseudoreferenced", and then pointerized, as this copies the semantics of unary & operator. One can easily understand why to do it when looking at the following source:

  ```
  int *i;
  int x;
  i = &x;
  ```

  In this case, the candidate function chosen for operator= (supposing that there was no user-defined operator= function and only those specified in **[13.6]** are taken into account) would be the one taking two arguments of int*, which is correct behavior.

- Probably the most complex of all the methods of sa_deconstruct_spse classes is visit_as_expression_function_call(). This function is triggered whenever there is operator () used in the source code. The occurrence of this operator can not only mean "plain" function call, but also things like overloaded operator() or function-style type cast must not be forgotten. First, arguments of the function call are handled recursively as usual. Then recursion on function prefix is also needed. The following cases can happen
  - the prefix is function-identifier (function name). Then we are in the most straightforward situation, and the function declaration has to be looked up using lookup and proper ss_funcall structure created. It is also possible that the lookup returns more than one function matching the name of prefix. Then overload resolution takes place, choosing the function to be called.
  - the prefix is name of object or structure. Then we are handling situation of overloaded operator () in the given object. Performing lookup in the class scope is needed, looking for the functions with name operator() and considering them as candidates to be passed to overload resolution. This is not yet implemented, but lookup is ready to perform the lookup task, however backend is not ready yet to handle calls to member functions.
  - the prefix can be name of a type. Then we are in function-style cast situation. Explicit user casts are not implemented yet. Implementing it would require just performing semantics check of type compatibilities according to the Standard, and then altering the type field in the expression.
  - the prefix is expression. There are quite a lot of cases to be handled - for example the expression can be dereference of pointer to function type. In this case, it is necessary to create ss_pfuncall or ss_ifuncall (see the documentation of ss structures for explanation of these structures), according to the type of the expression, etc. This is not implemented yet.

10.6.3 The Decision Between Builtin Operator and Function Call

Now we have to look in more detail what overloaded operator() of or_or_functional performs to understand the process of how the decision if builtin operator or function call will take place is being made (please note that this is called by the time when caller requests the creation of real ss_expressions, not sooner).

In case that overload resolution process had chosen the best viable function, the code of operator() of the returned functional creates and calls visitor or_declaration_helper, which is visitor on the ss_declaration type (which is type representing any semantic declaration). This is done because we need to know exactly what type of declaration are we currently processing - if it is a function declaration or builtin operator declaration (this can be recognized by visiting the ss_declaration, as both ss_function_declaration and ss_builtin_operator_declaration are derived from ss_declaration). In bodies of the visitor for all classes derived from ss_declaration, with exception for ss_builtin_operator_declaration and ss_function_declaration, the code immediately asserts, as it should not possible to reach this phase and do not hold either function or builtin operator declaration.

The creation of the actual ss_expression is handled by another visitor - because for different "groups" of operators, slightly different operations need to be taken.

For the case of the function call, the or_funcall_creator visitor is used (this is visitor on the ss_declaration type), which can then safely construct proper ss_funcall (ss_vfuncall, ss_mfuncall respectively), as the visitor is holding all the needed data (declaration and context). The decision if the function call in question to be generated is (any variant of) "plain" ss_funcall or if we are generating member function call (either method call or virtual function call, i.e. ss_mfuncall or ss_vfuncall respectively) can be done easily by checking the type of parent the the function declaration obtained from overload resolution - if the parenting scope is namespace declaration, we should generate plain function call. On the other hand, if the parenting scope is declaration of structure or object, we have to create member function call (or virtual member function call respectively, if the declaration of the function tells us that the function is declared virtual). The function call is created in quite a straightforward way. It is important to note, that there is a need to create new sequence point between function arguments, and also one sequence point before the function call itself, and one after it.

The or_builtin_operator_creator is triggered whenever the declaration of the created function call belongs to some builtin operator, and distinguishes the following different situations:

- The templated member function construct_bin_op_nosideeff_arith() is used to create those expressions which are binary arithmetic operators without sideeffect (for example binary +,-,*,/, etc.). This has to be handled in a different way than other binary operators, because backend requires, when handling

builtin arithmetic operations, to have all the argument types (and the return type) of the same type. Therefore this templated method first transforms subexpressions (which were propagated from upstream, from `or_or_functional`, which is holding them from the time of it's creation in overload resolution process) to the correct type (the return type of the function declaration that was chosen by overload resolution - which is, in case of arithmetic operators, the result of standard conversion sequence between parameters, as requested by **[13.6]**) and then calls operator `()` of the `or_or_functional`s representing subexpressions, thus resulting in creation of `ss_expression` and "wrapping" with conversion to correct type. This conversion, however, is not done if we are handling the case of pointer arithmetics - then the correct situation is to have *different* types for the arguments - the left one being pointer and the right one being `int` (or `ptrdiff_t` respectively).

- The templated member function `construct_bin_op_nosideeff()` behaves similarly to the previous one, but it is not performing type conversions for parameters to the type of the whole function. It just converts the types from pseudoreference (representing lvalues) to rvalues, using lvalue-to-rvalue conversion (which is represented by `ss_get` structure). This template covers for example relational operators (<, <=, !=, ==, >=, etc.).

- The templated member function `construct_bin_op_nosideeff_newsp()` handles operators, which have also the form of binary operation without sideeffect, as the the previous two, but there is need to create new sequence point between it's arguments. This means operators `||` and `&&`. The comma operator falls also into this category (binary op, no sideeffect, new sequence point has to be created), but is handled separately. This is because in the comma case, there is no need to create new expression, the result of the whole comma expression is the left subexpression (properly connected to sequence points). But in the case of `||` and `&&`, the actual expression, representing the operation, has to be created. Apart from this, the handling is the same as written in previous paragraphs for other binary operations.

- The templated member function `construct_un_op_nosideeff()` handles unary ops without sideeffect in quite a straightforward way - transforms the argument expression to the expected type and then constructs the corresponding descendant of `ss_expression`.

- The templated member function `construct_op_crement()` covers both prefix and postfix ++ and – operators. It is easy to distinguish if we are currently handling prefix or postfix operator - this information can be obtained by looking at the arguments in declaration, which was chosen by overload resolution. Then the assign operator (and corresponding new sideeffect) is created, to follow the semantics of the operators, defined by Standard (which means, besides other things, that in case of postfix increment/decrement, the return value is rvalue, but in the prefix case the return value is lvalue; in the postfix case the result of the whole expression is the add expression, but in the prefix case the result is the assign expression, being lvalue by definition).

  – If there is only one argument, we are handling the postfix case. In this case, we have to create new `ss_literal` structure, representing the integral value of '1', which will then be used in add expression. In case of arithmetics, this literal should have the same type as the left operand. In the case of pointer arithmetics, the literal has type int.

  – If there are two arguments, they are converted both to the same (left) type.

- The templated member function `construct_op_equal_sideeff()` is responsible for creating the `ss_assign` and `ss_{add,sub,mul,div,...}` pair of operations in the case of the +=, -=, ... operators. The handling is similar to the previous one, with the difference that we don't need to distinguish between prefix and postfix variants, and that the result is always lvalue.

- The templated member function `visit_ss_operator_assign()` creates `ss_assign` structure, representing assignment. For the same reason as described in previous paragraphs, we need to convert the right operand to the type of left operand, as backend is expecting this for all built-in types. Also the check on lvalueness of the left operand is done (the same as in previous operators, using assign). After creation of the `ss_assign`, sideeffect (which assignment alway has) is created and connected properly to the structure.

10.6.4 Return Value

In both cases described in previous section, expression of type `ss_expression` was created. This is then returned through the chain of visitors (each of them contains the field `result` of `ss_expression` type, designed to return

the value), through `sa_deconstruct_spse` to the original caller, who wanted to create expression according to the type of `as_expression`.

10.7 Overload resolution

10.7.1 Overload resolution overview

Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be arguments of the call and set of candidate functions that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the types of the parameters of the candidate function, how well (for non-static member functions) the object matches the implied object parameter, and certain other properties of the candidate function. Overload resolution performs the selection in different contexts within the language. These are namely:

- **[13.3.1.1.1]** invocation of function named in function call syntax. This is the most usual case of calling functions "directly".
- **[13.3.1.1.2]** invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax.
- **[13.3.1.2]** invocation of the operator referenced in an expression.
- **[13.3.1.3]** invocation of constructor for direct-initialization **[8.5]** of a class object
- **[13.3.1.4]** invocation of a user-defined conversion for copy-initialization **[8.5]** of a class object.
- **[13.3.1.5]** invocation of a conversion function for initialization of an object of a non-class type from an expression of class type.
- **[13.3.1.6]** invocation of a conversion function for conversion to an lvalue to which a reference will be directly bound.

In the following sections, the implementation of overload resolution and other parts connected with this process (such as implicit conversion sequence finding), is described.

10.7.2 File Structure

Overload resolution, being part of semantic analysis, is implemented in `lestes/lang/cplus/sem` directory, in files whose names start with or_* prefix. First, contents of these files will be roughly summarized, to give better overview.

- **or.cc** - this file contains implementation of the 'high level' overload resolution algorithm, as described in **[13.3].** In this file candidate selection, viable function and then best function selection is done. This file also contains machinery, which is used to create special set of declarations, as specified in **[13.6]**, containing declarations for all builtin operators, which are then used by overload resolution as the subset of candidate functions.
- **or_ics.cc** - this file contains implementation of algorithms and functions which are related to implicit conversion sequences (sequences converting one type to another). Right now only subset of conversions, defined by C++ standard is supported - conversions between builtin types, as described in chapter **4**, with exception for **[4.11/1]** and all paragraphs concerning qualification conversions (namely **[4.4]** and **[4.10]**) and derived-to-base conversion. This is special conversion, used solely for purposes of overload resolution and is specified in **[13.3.3.1/6].**
- **or_ics.hh** - this file contains exported interfaces, which are meant to be used by users of **or_ics.cc** file, namely function `or_find_ics()` implemented in this file. This file also contains helper macros and templates, which are used to fill in special decl_seq, holding all the declarations as specified in **[13.6]**, representing declarations of builtin operators.

- **or_or.cc** - this file contains some helpers for purposes of the overload resolution process. Namely it contains visitor on `ss_declaration`, which is needed to determine, if we are currently working with builtin operator, conversion function or ordinary function. This file also contains overloaded `operator()` of functional, which is used to construct `ss_expression` of the proper function call. The purpose and functionality of implementations done in this file were already discussed in 10.6.3.

10.7.3 Programming Technique Used—Functionals

In previous paragraph, the term **functional** was used. The idea behind functional is quite simple - functional is a class/object, which has overloaded `operator()`. When designing algorithm for finding implicit conversions, it turned out that straightforward constructing of the data structures in memory would become very expensive in terms of memory consumption, as the algorithm works basically as BFS search. A need for object which could be simply cached and which would also serve purpose of good readability (i.e. the whole process will appear to the user of the interface just as calling ordinary function) was satisfied by **functional**. We will show on example of conversion sequences how functionals could be used to prevent memory consumption being too vast.

First, there are five types of functionals in our situation (see **or_ics.lsd**)

- functional for standard conversion
- functional for conversion from lvalue to rvalue
- functional for conversion by constructor
- functional for conversion by conversion function
- functional for compound conversion

Each of these functionals represents the conversion **to** given type (for example functional for standard conversion can represent conversion to `signed int`, another functional for standard conversion can represent conversion to `boolean`, functional for conversion by constructor can represent copy-initialization by one given constructor, etc). Note that the functional is not dependent on the **source** type of the conversion - for all conversions to `signed int` (for example), there is need to create just one functional, representing all such conversions. The source expression, from which the conversion is done, is passed to the functional (or, to be more precise, to the object's overloaded `operator()`) right at the time when `ss_expression`, representing the conversion sequence, has to be constructed.

A good example of this technique is used in the functional implementation itself - in functional for compound conversion, to be more precise. This functional contains, as it's fields, pointers to other functionals - *outer* and *inner* conversion. The result of the functional is then simply obtained by calling *outer* functional on the result of *inner* functional for given argument:

```
ptr< ss_expression > or_ics_functional_for_compound_conversion::operator()
                                            (ptr<ss_expression > arg) {
                     return (*outter_conversion)((*inner_conversion)(arg));
}
```

(this code comes from **or_ics.cc** and is the implementation of `operator()` for **or_ics_functional** representing compound conversion).

Functionals for other types of conversions, apart from compound, are implemented in the same file, and they main task is to create semantic structure for given action (function call for constructor/conversion function, `ss_get` for getting rvalue (i.e. performing the lvalue->rvalue conversion), `ss_conversion` for standard type conversion, etc.) and returning the resulting `ss_expression` (usually consisting of more than one conversion and "merged" together by functional for compound conversion, mentioned above) to the caller, who expects `ss_expression` representing the given conversion sequence as semantic expression.

10.7.4 Implicit Conversion Sequences

An implicit conversion sequence is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. Implicit conversion sequences are concerned only with type, const-volatile qualification and lvalueness of the argument expression and how these arguments can be converted to match the corresponding properties of the parameter from the function prototype.

The main "engine" of the part, which finds implicit sequence, is the function *or_find_ics()* with the following prototype

```
ptr< or_ics_functional > or_find_ics(ptr< or_or_functional > source,
                                          ptr< ss_type > target);
```

This function is used to find implicit conversion sequence from expression *source* (the parameter is functional, described in 10.6 - in short, it is "creator" of the `ss_expression` and as such holds all the information about the expression, which finding implicit conversion sequence requires) to the type of *target*. To achieve this, we need to know how finding implicit conversion sequence works.

Implicit conversion sequence is a sequence consisting of *left standard conversion sequence*, followed by *user conversion sequence*, followed by *right standard conversion sequence*. For definition of terms *standard conversion sequence* and *user conversion sequence*, see **[4], [13.3.3.1.1]** and **[13.3.3.1.2]**. What must be explicitly noted here is, that **[13.3.3.1/6]** teaches us about derived-to-base conversion (conversion of derived class to any of it's ancestors), which is for purposes of overload resolution, considered **standard** conversion sequence, having **conversion rank**.

Let us now look at possible forms of implicit conversion sequences

- If the *source* is of user-defined type (i.e class), the *left standard conversion sequence* can be identity or derived-to-base conversion (or can be omitted). Then **user-defined conversion**, using conversion function, can be performed, to convert this user type to another builtin type (note that in implicit conversion sequence, at most one user-defined conversion can take place), which can be followed bi *right standard conversion sequence*, converting from the result of user-defined conversion to the target type
- If the *source* is of builtin type, the *left standard conversion sequence* can be constructed by following rules in chapter **4** (converting builtin type to another builtin type), followed by **user-defined conversion** (copy-initialization using copy constructor), followed by *right standard conversion sequence*, which can be either identity, or **derived-to-base** conversion.

Function `or_find_ics()` performs BFS search, generating all possible conversions from `source` and filtering out those not having the target type of `target`. Currently, the BFS constructs only **standard** conversion sequences from `source` to `target`, as user-defined conversions are not supported yet, but they can be easily added to the code - the further level of BFS, performing user defined conversions and later *right standard conversion sequence*, can be inserted into `or_find_ics()` function.

```
ptr< or_ics_functional > or_find_ics(ptr< ss_expression > source,
                                     ptr< ss_type > target) {
[1]        scs_imaginable_tgts = source->type_get()->accept_or_ics_base(v_t);

[2]        filtered_list = filter_type(scs_imaginable_tgts, target);
[3]        best_conv = get_best_conversion(filtered_list, source);
           return best_conv;
}
```

The above code snippet is simplified version of `or_find_ics()` function, which can be found in **or_ics.cc** file. The line marked [1] calls visitor, working on top of `ss_type`, which, depending on the dynamic type of `source->type_get()` (representing the *real* source type), constructs all possible standard conversions from this type. The visitors for all rvalue types are quite simple - they just create functional for every possible conversion from the given type and put them into the list, which is later returned to the caller. The types which are worth of explicit discussion are:

- `ss_class` - the only standard conversion defined for this type is derived-to-base conversion, as was discussed previously in this chapter. What the visitor, when hitting this type, does, is that it inserts into the list, which is later returned to the caller, functionals representing conversions to all the possible bases for given class. All of them have the same *conversion* rank.
- `ss_pointer` - the Standard prescribes conversions that can be done with pointers in **[4.10]** and **[4.11]**. The only conversion currently implemented for pointers is the pointer-boolean conversion, but other conversions can be easily added to the `visit_ss_pointer` and `visit_ss_member_pointer` method of the `or_ics_visitor_tgts` visitor.
- `ss_reference` - for the reference type, identity conversion and lvalue-to-rvalue conversions are defined. The first one is needed to be able to assign one reference to another reference (if the types match), the second one is needed to be able to write constructs such as

```
int a;
int b;
int &z = b;
a = z; [A]
```

The lvalue-to-rvalue conversion takes place on the line marked `[A]` as z is reference type and to be able to store it into lvalue variable of type `int`, the right side of assignment has to be converted to rvalue (as can be easily seen from builtin operator declaration for assignment operator in **[13.6]**).

- `ss_pseudoreference` - pseudoreferences is type used in lestes compiler to represent any lvalue, which is not reference. This means that anything having this type can be converted to rvalue of any such type, to which the underlying type of the `ss_pseudoreference` can be implicitly converted. Therefore, when performing the BFS search in the `visit_ss_pseudoreference()` method, lvalue-to-rvalue conversion is generated as one of the possibilities. Then the result of this conversion is converted by recursive use of the visitor to all types, to which the rvalue can implicitly be converted. The next notable conversion, which is then generated in this method, is reference-binding conversion. This conversion takes place in two cases. The first one is calling the function which is having argument of reference type, with pseudoreference (i.e. lvalue), argument. Let's have the following example

```
int f(int &a)
int main()
{
int z = 1;
f(z);
}
```

Here, when the function call is performed, the reference binding conversion has to take place for parameter `z`, as the variable itself is lvalue, but it has to be converted to the reference type. Very similar situation happens, when the initialization of variable is done, such as

```
int i;
int &z = i;
```

In this case, the reference binding conversion also takes place, as the lvalue `int i` has to be converted to the target type of reference `int &z`.

The line [2] filters out from this list conversions not having the target type of the same as `target`, and `get_best_conv()` function gets from the list of conversions the best one, which is then returned to caller (or, to be more precise, the *functional* representing this conversion is returned).

`get_best_conv()` function is implementing rules for comparing the "goodness" of conversion sequences. The ranking of conversion sequences is described in **[13.3.3.2]**. Just a short note: because of member pointer conversions (which "go" from parent classes to derived classes), the `ss_class` structure, representing user defined class, has to contain the list of it's successors. The relation "to be better conversion" is defined as follows.

Notation:

- $\mathbf{A} \lhd \mathbf{B} \Longleftrightarrow \mathbf{A}$ is direct ancestor of **B**
- $\mathbf{A} \unlhd \mathbf{B} \Longleftrightarrow \mathbf{A}$ is ancestor of **B**
- **UCS** is **user conversion sequence**
- **SCS** is **standard conversion sequence**
- **rank(x)** is **0** for exact, **1** for promotion, **2** for conversion, **3** for user
- $u_i \equiv l_i c_i r_i$, where l_i, $r_i \in$**SCS,** $_i \in$user conversion function, $u_i \in$**UCS**
- $s_i \in$**SCS**

Definition:

- $\mathbf{A} \unlhd \mathbf{B} \Longleftrightarrow_{\mathbf{def}} (\mathbf{A} \lhd \mathbf{B}) \vee (\exists \mathbf{C})\ (\mathbf{A} \lhd \mathbf{C} \unlhd \mathbf{B})$

Definition (This definition defines relation $u_1 \succ_c u_2$, which is read such as "u_1 is better conversion than u2"):

- $u_1 \succ_c u_2 \Longleftrightarrow_{def} [(c_1 = c_2) \wedge r_1 \succ_c r_2]$
- **[13.3.3.2/3]** $s_1 \succ_c s_2 \Longleftrightarrow_{def}$
  $[(\zeta_1$ is subsequence of $\zeta_2) \wedge \zeta_i$ is s_i without lvalue transformation$)) \vee$
  $\vee\ (\text{rank}(s_1) < \text{rank}\ (s_2)) \vee$
  $\vee\ \{((s_1 \setminus \text{qual}(s_1)) = (s_2 \setminus \text{qual}(s_2)) \wedge \text{target}(s_1) \approx \text{target}(s_2) \wedge \text{cv_sig}(\text{target}(s_1) \subsetneq \text{cv_sig}(\text{target}(s_2)) \wedge$
  [4.4] $s_1 \neq$ (string literal $\hookrightarrow$ char *) **[4.2]**$\} \vee$
  $\vee\{\text{is_ref_bind}(s_1) \wedge \text{is_ref_bind}(s_2) \wedge \text{cv_strip}((\text{target}(s_1)) = \text{cv_strip}((\text{target}(s_2)) \wedge \text{cv_extract}(\text{target}(s_1))$
  $\subsetneq \text{cv_extract}(\text{target}(s_2))\} \vee$
  $\vee(\text{rank}(s_1) = \text{rank}(s_2)) \wedge$
  $\wedge\ \{(s_1 \neq ((\text{pointer} \mid \text{member_pointer}) \hookrightarrow \text{bool}) \wedge s_2 = ((\text{pointer} \mid \text{member_pointer}) \hookrightarrow \text{bool})) \vee$
  $\vee(A \trianglelefteq B \wedge (s_1 = (B^* \hookrightarrow A^*) \wedge s_2 = (B^* \hookrightarrow \text{void}^*)) \vee$
  $\vee(A \trianglelefteq B \wedge (s_1 = (A^* \hookrightarrow \text{void}^*) \wedge s_2 = (B^* \hookrightarrow \text{void}^*)) \vee$
  $\vee(A \trianglelefteq B \trianglelefteq C \wedge [(s_1 = (C^* \hookrightarrow B^*) \wedge s_2 = (C^* \hookrightarrow A^*)) \vee$
  $\vee(s_1 = (C \hookrightarrow B\&) \wedge s_2 = (C \hookrightarrow A\&)) \vee$
  $\vee(s_1 = (A::^* \hookrightarrow B::^*) \wedge s_2 = (A::^* \hookrightarrow C::^*)) \vee$
  $\vee(s_1 = (C \hookrightarrow B) \wedge s_2 = (C \hookrightarrow A)) \vee$
  $\vee(s_1 = (B^* \hookrightarrow A^*) \wedge s_2 = (C^* \hookrightarrow A^*)) \vee$
  $\vee(s_1 = (B \hookrightarrow A\&) \wedge s_2 = (C \hookrightarrow A\&)) \vee$
  $\vee(s_1 = (B::^* \hookrightarrow C::^*) \wedge s_2 = (A::^* \hookrightarrow C::^*)) \vee$
  $\vee(s_1 = (B \hookrightarrow A) \wedge s_2 = (C \hookrightarrow A))\}])$

  The majority of these relations are implemented in `is_better_conversion_seq()`.

10.7.5 Overload Resolution Itself

Programming overload resolution on top of what was described above is relatively easy. The main function, performing this task, is *overload_resolution()*, defined in **or.cc** with the following two alternative prototypes

```
ptr< or_or_functional > overload_resolution (ptr< exprlist > exprs,
                                             ptr< as_name > func_name)
ptr< or_or_functional > overload_resolution (ptr< exprlist > exprs,
                                             ptr< func_decl_set > candidates)
```

These two functions differ in the way how they collect functions, which are later considered as candidates for viable function during overload resolution process.

The first function obtains a function name and performs lookup, provided by lookup interface implementation**.** The lookup is performed in the current scope (this can be obtained from `sa_context_manager`) and also in the special scope, containing all declarations for builtin operators, as defined in **[13.6].** This scope is represented as singleton - during the first invocation, the scope is created and filled with declarations, and all subsequent uses of the decl_seq just return the already filled structure. This is implemented in the `or_builtin_operator_declaration_creator::instance()` and makes use of macros and templated functions declared and defined in file **or.hh**.

The second one, however, gets the candidates from "upstream" directly as declarations set. This interface is called from deconstruct SPSE process, described in 10.6, when handling the abstract-syntax expression for function call, to perform overload resolution on the prefix. By the time deconstruct process is doing this, it already has set of candidate declarations.

Then selection of viable candidates is performed, which means mainly filtering the functions with "correct" number of arguments and for which implicit conversion sequence between their parameters and expressions, representing parameters in call, exists (this makes use of `or_find_ics()` function, documented above).

When the list of viable candidate functions is obtained, the *best selection* is performed, to select the best function for overloading. For the definition of *best* function, see below and **[13.3.3].** Implementation of this relations described in this chapter in Standard, is contained in function `best_selection()`, which heavily uses `is_better_function()`, where rules for comparing functions, as written in **[13.3.3],** are implemented.

As the compiler currently doesn't support templates, we have omitted comparisons related to template functions. But once the template support is added, the needed checks, as written in **[13.3.3],** have to be added to the function `is_better_function()`, to compare template functions properly in overload resolution. The relation "to be better function" is defined as follows:

- $\mathrm{ICS}_\mathrm{i}(\mathrm{F})$ = {*converting_expression_constructing_program, rank* } (this denotes conversion sequence for i-th parameter in function call between argument type and parameter type).
- $\mathrm{ICS}_\mathrm{i}(\mathrm{F}) \succ_c \mathrm{ICS}_j(\mathrm{G})$ if “better conversion”, see **[13.3.3.1]**
- $\mathrm{F} \succ_f \mathrm{G} \Longleftrightarrow {}_{\mathrm{def}}[\ (\forall \mathrm{i};\ \mathrm{ICS}_\mathrm{i}(\mathrm{F}) \succeq_c \mathrm{ICS}_\mathrm{i}(\mathrm{G})] \wedge$
  $\wedge\{[\ \exists \mathrm{j};\ \mathrm{ICS}_\mathrm{i}(\mathrm{F}) \succ_c \mathrm{ICS}_\mathrm{i}(\mathrm{G})\ ] \vee$
  ∨ [F is not template ∧ G is instantion of template specialisation] ∨
  ∨ [both F and G are templates ∧ $\mathrm{F} \succ_{templspec} \mathrm{G}$ **[14.5.5.2]**] ∨
  ∨ [context is initialization ∧ SCS (rettype(F) → dest) $\succ_c$ SCS(rettype(G) →dest)]}

After the best function is chosen, functional representing it is returned to the caller.

10.8 Work To Be Done

- conversions by constructor and conversion functions
- using directives and using declarations
- when performing pointer arithmetics, not all combinations work yet (namely pointer + non-rvalue).
- member pointer conversions and proper ranking of reference binding conversions
- multi-level pointers conversions and qualification conversions, as specified in **[4.4]**
- adjusting the process of finding candidate functions according to **[13.6],** to support wider variety of builtin functions it is able to find. This involves particularly member access operators, pointers to user-defined types, pointers to `const-volatile` qualified data, etc.
- as soon as templates are implemented, implement choosing the best viable function also for templated functions.
- add support in Deconstruct SPSE process for transforming wider set of expressions (member access, `new`, `delete`).

Chapter 11

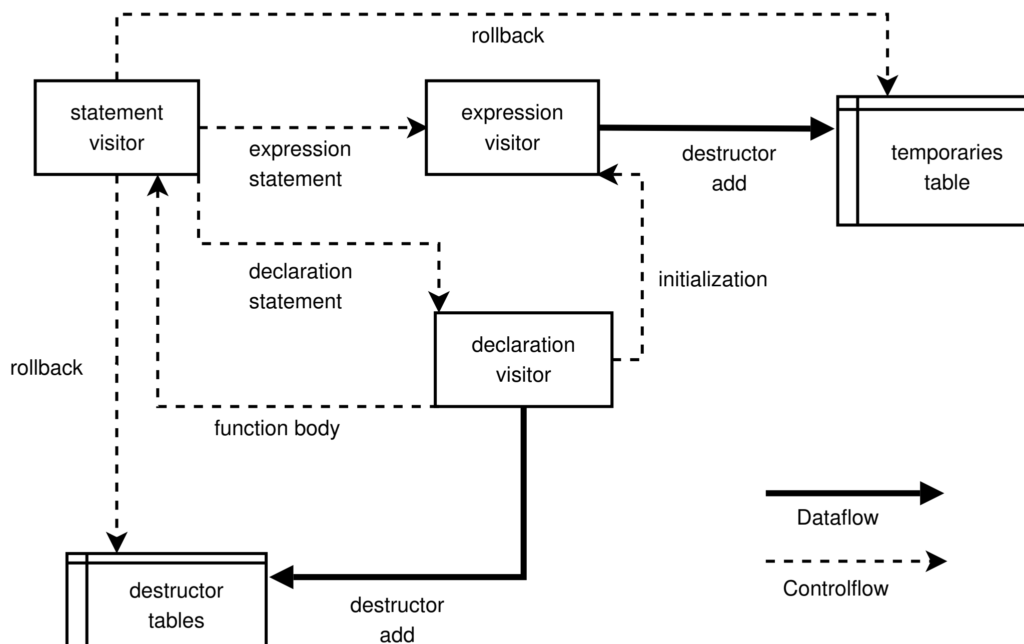
Transformation of SS to PI (ss2pi)

11.1 Introduction

The ss2pi transformation is responsible for transformation of semantic structures to backend structures. The most C++ related information is lost during this conversion to maintain a backend independent on the translated language. That means that all expressions, statements and declarations are translated into pseudoinstructions, labels and literals.

These transformations are implemented via *visitor* design pattern. Each category of objects—expression, statement and declaration—has corresponding visitor governing transformation. C++ types have to be also converted into their backend equivalents. This conversion, however, is target dependent and thus only backend interface shall be called.

The basic overview of ss2pi modules can be seen in the following picture.



The only thing that pervades from the source code is the structural info describing rules of data&control flow. This is the key property for future development of optimizations in pseudoinstructions linearization. This is done by copying of *sequence point graph* (SPG) to backend—sequence points give rules for ordering side-effects with different psp&nsp, while the order of evaluation and side-effects with the same psp&nsp is still undetermined and is fully governed by the backend.

11.2 Supporting structures & algorithms

Before description of individual transformations, we describe supporting structures, which are needed for such conversions.

11.2.1 Sequence points transformation

With two exceptions, this transformation simply means copy of the SPG from *ss* layer to *pi* layer, so the structure remains the same.

Exceptions are:

- Converted are only sequence points, which are actually used for evaluation of expression. For example, for expression in the following code `int a,b,c; (a+1,b),c=1;` we have one SP between `a+1,b,` but as there is no sideeffect in the whole parentheses, there will be no code generated. Hence the SP between `a+1,b` will not be reached and transformed.
- SPs are added for destructors of objects. In case of temporary objects, SP will be simply added after the full-expression (as **[12.2/3]** requests destructors there). In case of local variables SPs for destructors are added :
  - before jump (goto, break, continue, return)
  - after destructor-SP (end of compound statement)
  - for global initializations/finalizations of the translation unit

11.2.1.1 Implementation

Transformation is done by *ss_sp2pi_sp* function. It is a lazy function in the sense, that if a given SP *x* is

- already converted, then only returns converted pi_sp
- not converted, then creates corresponding pi_sp and recursively converts *x.psp* and *x.nsp* .

Each creation has to be announced to backend by *backend_data_builder->add_sp* API.

11.2.2 Destructor tables

Because **[6.6/2, 12.2/4]** requires proper destruction of variables, we need to know for each point of program, which variables are to be destructed, shall the compound statement be abandoned.

Second is the case of temporary variables, which are created during evaluation of expression and have to be destroyed at the end of their containing full-expression **[12.2/3]**.

11.2.2.1 Structure of destructor tables

- each entry for destruction of variable consists of
  - variable to be destroyed
  - destructor called for destructing the variable (variable itself is not enough, because it is context dependent, which destruction is to be called—e.g. local vs. global variables)
- list of entries forms one *destructor table.*
  - each expression has associated one *destructor table* for temporaries.
  - each compound statement has associated one *destructor table* for variables declared in this compound statement (excluding variables in nested compound statements)
  - API: *destructor table* itself has method *back_roll_table* for "back-rolling" of its variables, which means, that it generates the code for proper destruction of all the variables in the table.
- structure of destructor tables copies structure of compound statements and builds up *destructor tables* object.
  - there is only one *destructor tables* object for a given translation unit.
  - API: For back-rolling variables from nested compound statement to some of its ancestors *back_roll_to_compound_stmt* is defined. This method is consequently used for back-rolling at the end of current compound statement (*back_roll_from_current_compound_stmt*) and back-roll for the whole function in case of end or return statement (*back_roll_from_function*).

11.2.2.2 Implementation

Class *ss_destructor_tables_stack* implements *destructor tables*. *Destructor table*-s are in a list (used as a stack), copying the nesting of compound statements, which we are currently transforming. The last (*current*) *destructor table* is appropriately pushed/popped in/from the list as we are entering or getting out of the compound statement.

However this stack-implementation will not be enough for goto statement implementation. For its solution see 11.3.8.

Class *ss_destructor_table* implements *destructor table* object. Entries in this table are also organized into a list, as arrays require destruction of elements in an order reverse to the order of their creation.

For accessing proper constructor/destructor for a given type API of *ss_constructor/destructor_finder* class is used. Up to know frontend does not implements this interface, as classes are not implemented fully.

11.2.3 Variable maps

Variable map provides map from variable to corresponding *pi_mem_factory* object (which in backend represents memory placeholder and is equivalent to lvalue in *ss* layer).

We distinguish between

- namespace-scope variables—these are variables contained in any namespace scope including global scope and unnamed namespaces. These are considered one category for the purpose of backend.
- local variables—variables declared inside function body.

11.2.3.1 Implementation

Maps are implemented through standard STL maps.

There are two kinds of map—the first map of namespace variables for the whole translation unit, the second, for local variables, is created separately for each function body.

Distinguishing what kind of map is to be used for a given variable is under statement transformation visitor liability 11.3.

11.3 Statements transformation

Visitor *ss_stmt2pi* is responsible for statement conversion. Each visitor contains handle of the current function, which is transformed, and variable maps, which are accessed through *variable_mem_get/set* method. For details see doxygen documentation.

Now the individual statements:

11.3.1 Compound statement

This is the basic block, from where all the transformations begin. As every compound statement is paired with a corresponding declaration sequence, it is possible to run the transformation by passing either compound statements or declaration sequences. We decided for compound statements, so the whole transformation from start to end walks through compound statements and no declaration sequences are being visited.

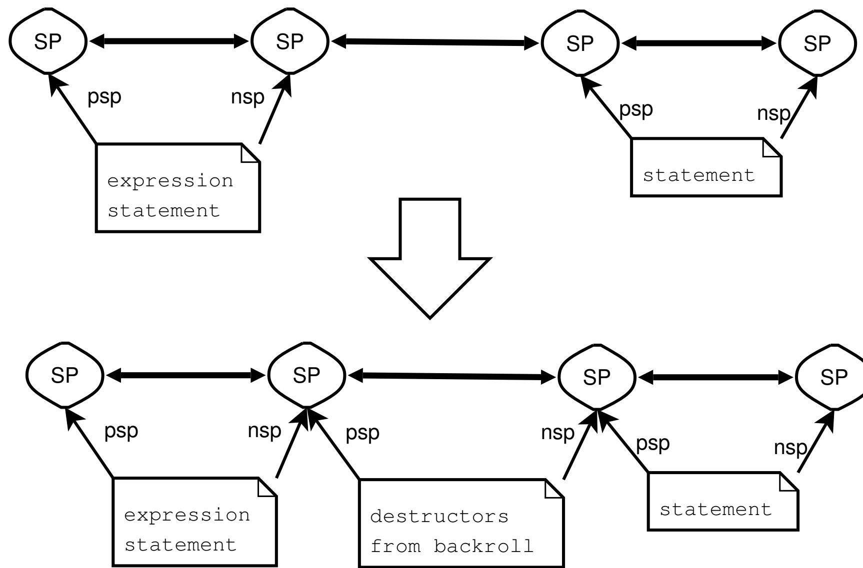
Details: New statement visitor and new destructor table is created for each compound statement. When all statements inside are visited (i.e. transformed), all the variables from current compound are back-rolled after destructor sequence point(*dsp*) at the end of compound statement.

11.3.2 Declaration statement

In this case, control is simply transferred to declaration visitor. See 11.4.

11.3.3 Expression statement

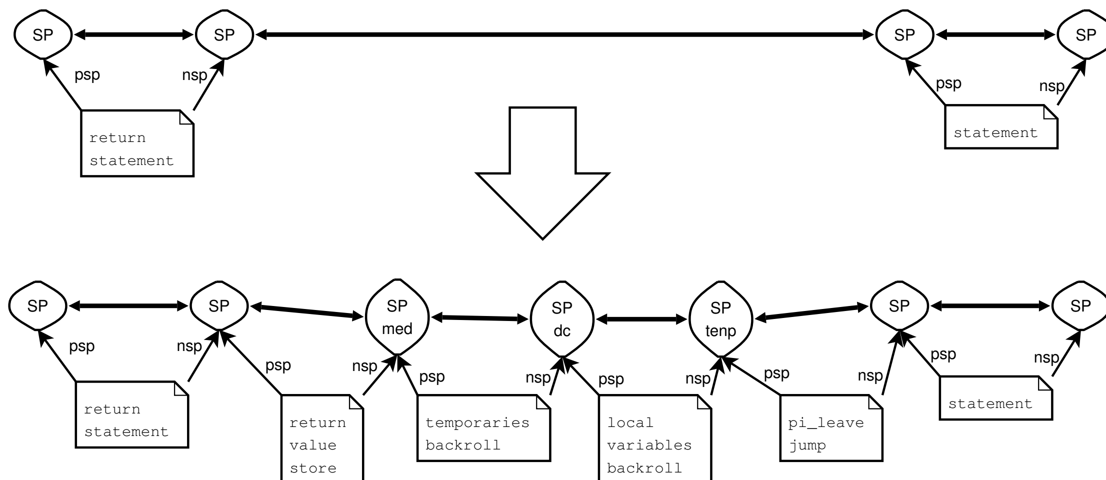
Before any expression is transformed we have to create temporaries table for objects created during evaluation of expression and give correct arguments—that this is widely commented in 11.5 expression transformation. One expression statement represents full-expression and therefore back-roll of temporary objects will be called after. The situation can be seen on the picture.



Evaluation itself does not start from the whole full-expression, but from side-effects, which only have to be performed and only subexpressions which are needed for such a performance are evaluated. Proper ordering of side-effects and evaluations is guaranteed by sequence points, which are passed to backend.

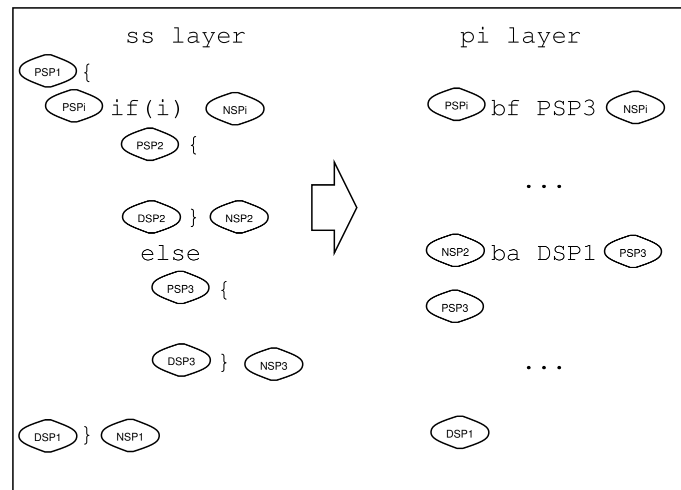
11.3.4 Return statement

This case is more difficult, because not only temporaries from expression have to be destroyed before jump, but also back-roll of appropriate local variables has to be done. The transformation can be seen on the picture.



11.3.5 If statement

If statement is done using two branch instructions. Transformation can be seen in the following picture.



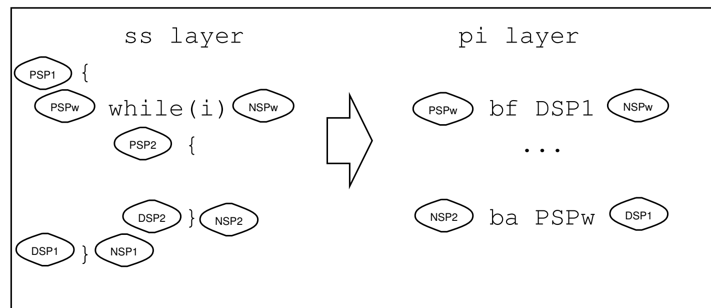
Condition is transformed via expression visitor and compound statements are transformed using statement visitor.

Note, that *ba* branch instruction shall jump to *DSP1*, to destruct created object in case of `if (class_type A =..)`.

For detailed description of *if statement* on ss layer see 10.4.9.

11.3.6 While statement

While statement transformation is similar to if statement. For detailed description of *while* on ss layer see 10.4.10 .



11.3.7 Other loop statements

Other loop statements(do, for, break, label) are to be treated similarly, and are not implemented yet. For continue and label statements, appropriate sequence points are available and also code for backroll of current scope variables is implemented.

11.3.8 Goto statement

To transformation goto statement we have to know labels inside the currently transformed function. To provide it, we have to collect all the labels inside the function, which can be done

1. in *as* -> *ss* transformation
2. in ss2pi transformation

Transformation of goto statement cannot be solved in the first phase of *ss2pi*.

Rationale:

In 1. destructors for locally introduced variables of the scopes, which we are leaving, have to be called. It is not clear which have to be called.

In 2. case we even don't know where to jump.

Thus, for transforming goto statement we need second pass of ss2pi algorithm, in which it will be known proper destructors and jumping positions.

11.3.8.1 Destructor trees

We introduce *Destructor tree*, which has to be done in the first pass, in order to manage whole transformation in the second pass.

Now the construction

Active object is an object that is in a scope or that would be in a scope provided that it is not hidden by another active object declaration. Example:

```
{
  int i;
   {
     int i; // here both i's are active, despite the first is hidden by the second.
   }
}
```

Destructor tree (*DT*) is a structure representing active objects for any given point in a function. It consists of nodes, each of them having a parent.

The path from node to the farthest ancestor represents a sequence of active object at a given point in the function. In addition to that it also represents a compound statement boundaries in a manner that can be used to distinguish to which compound statement the active object belong. It can be seen, that this structure is tree-like.

Having constructed DT, we can index the corresponding node in DT by the statement start. Now to transform a goto statement consists of:

1. To find the nearest common ancestor(*NCA*) of the goto statement and goto-statement-target (statement corresponding to the referred label)

2. To generate destructor call for active object on the path from goto statement to NCA.

3. To check, whether all the active object from target to NCA are non_initialized POD types [].

4. To allocate all the variables from target to NCA.

5. To generate pseudoinstruction of jump to target.

Goto statements are not implemented now. To construct DT, implementation of destructor tables(11.2.2) have to be changed from single LIFO to tree-structure, however current interface for destructor table is enough and don't need any change.

11.4 Declarations transformation

Visitor ss_decl2pi is responsible for conversion of declarations.

11.4.1 Object declaration

This declaration represents object in the terms of Standard **[1.8/1]**. It represents memory placeholder and it is represented by *pi_mem(factory)* class in backend.

11.4.1.1 Local-scope objects

Local scope objects are declared inside function. At the point of declaration proper *pi_mem_factory* has to be allocated in backend via *mem_alloc_manager* API. Then object has to be registered in variable maps*11.2.3.* Finally proper destructor shall be added to destructor tables.

Up to now only copy initializations[1] are supported.

[1]Types of initializations are: copy initialization (int i=3; int i=f(x);), direct (int i(3);), default (int i;), zero (namespace {int i}), default initialization (int i). For details see doxygen documentation for *ss_decl_stmt.initializer_kind*

11.4.1.2 Namespace-scope objects

Namespace-scope objects are to be treated differently. As described in ABI, special register functions are to be called for their initialization. These initializations are not implemented. Only simple initializations by literals are supported now, as they do not require call of any function. Also note, that any initialization requiring either conversion or function call being performed do not work yet.

Up to now only unnamed namespace(global scope) is supported. Backend itself does not distinguish namespaces, so the future implementation shall put the *ss* objects representing variables in all namespaces in one specialized list, which will be processed by ss2pi.

11.4.1.3 Look-up

There exists a special visitor *ss_decl2mem* which looks up proper backend memory placeholder (*pi_mem_factory*) for a given object declaration which involves correct care of function parameters and fake declarations. This visitor is used by variable maps.

11.4.2 Function declaration

The whole function is transformed via this declaration.

- Details: Proper visitors and variable map for objects in the function are created. Function parameters are processed by backend itself, as their order and size is target depended.

11.4.3 Structure declaration

In this case VMT table for given structure shall be created. Not implemented now.

11.5 Expressions transformation

For each full-expression new *ss_expr2pi* expression visitor is created. We shall describe common attributes for expression visitors.

Generally expression transformation visitor firstly visits operands of a given (sub)expression and take their results. After that converts subexpression itself to corresponding pseudoinstruction and returns subexpression result (i.e. destination of transformed pseudoinstruction).

There are cases[2] , where the resulting value of subexpression is needed more times by visitor, while the code for evaluation shall be generated only once. So, every expression visitor shall firstly check, whether current semantic expression was evaluated and in such a case just return resulting operand via *ss_expr2pi.expr2operand_map*.

While recursively evaluating subexpressions, sub-results can be of two types—memory placeholder (*pi_mem*) or value (*pi_preg*[3]), this corresponds to lvalue-rvalue distinction. In *ss* intercode this distinction is reflected via the type of expression, i.e. expression is lvalue ⇔its type is descendant from *ss_referential*. Types of propagated subexpression results can be seen in the picture 11.5.2(Pointers, references and l-values).

In case of accessing an object in memory we use a placeholder for the memory location (*pi_mem_factory*). The placeholder represents the memory location irrespective of the actual state of the memory. We have to obtain a properly timestamped *pi_mem* object, representing the memory location together with its state. Roughly it can be said, that *pi_mem_factory* represents memory region and *pi_mem* is that region in a certain time. In order to provide a possibility for optimizing out certain memory accesses the *pi_mem* object has to hold a reference to the last visible store to that memory location. Two independent loads of the same pi_mem can be represented by only a single *pi_ld* pseudoinstruction.[4] The location of the last store is computed by *ss_expr2pi.compute_origin* method. When there is no previous access in the current step of transformation of a given full-expression, the starting sequence point of the whole full-expression (*ss_expr2pi.psp*) is used as the conceived store location. Other information about *pi_mem* and *pi_mem_factory* object can be found in backend description (see 6.3.1).

For correct working of the ordering algorithm, each pseudoinstruction has a level information associated, which is computed as maximum of its source operand & sequence point levels increased by one. The exact meaning and operational semantics of the levels in the ordering process is discussed in **??**.

[2]e.g. `i++` needs two times var-ref access, or multiple access of result when *is_returned_pointer* is used

[3]both *pi_preg* and *pi_mem* are descendants of *pi_operand*.

[4]This is not implemented yet.

Destructors for temporary variables used while evaluating the full-expression are in *ss_expr2pi.temporaries* destructor table.

For detailed information about all the attributes of *ss_expr2pi* visitor see doxygen documentation.

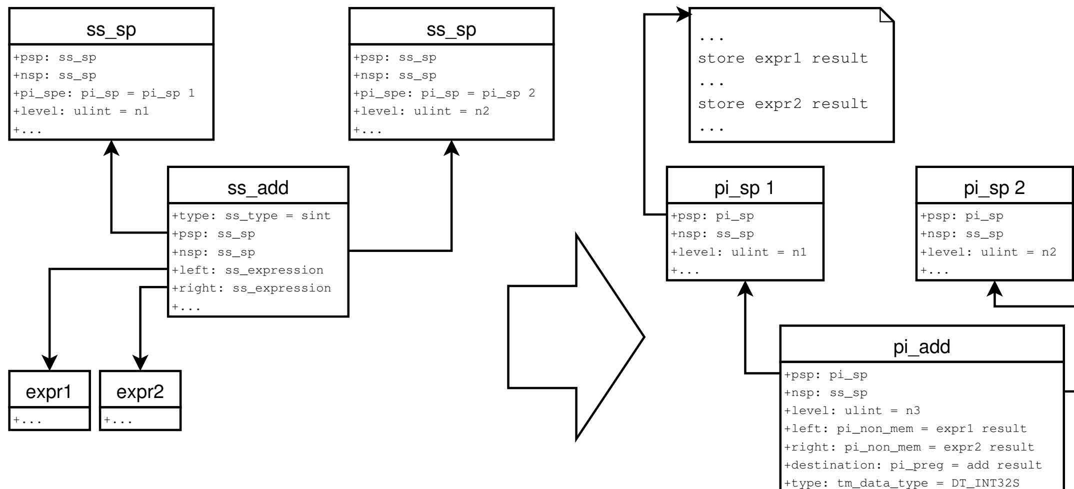
Now we describe particular visitor methods.

11.5.1 Binary operators

We take the binary operators as an exemplary conversion and describe things which hold also for another expressions and will not be explicitly remembered again.

As far as transformation is concerned, most binary operators are similar[5]. First, left and right subexpressions have to be transformed and a pseudoregister for the result has to be created. The target type of the pseudoregister is determined by backend *ss_type_to_tm_type_convertor.* Then the pseudoinstruction corresponding to the binary operator, with appropriate sequence points, operands, level and destination is created. To let the backend know about currently created pseudoinstruction, we call *backend_data_builder.add_pi*, which is responsible for proper insertion of the pseudoinstruction into backend structures. Also note, that a part of the current expression transformation can also be conversion of previous/next sequence points, which can be transitively accessible from current expression, in case these sequence points were not transformed into backend version of *SPG* yet.

One example of such a conversion can be seen at the picture.



11.5.2 Pointers, references and lvalues

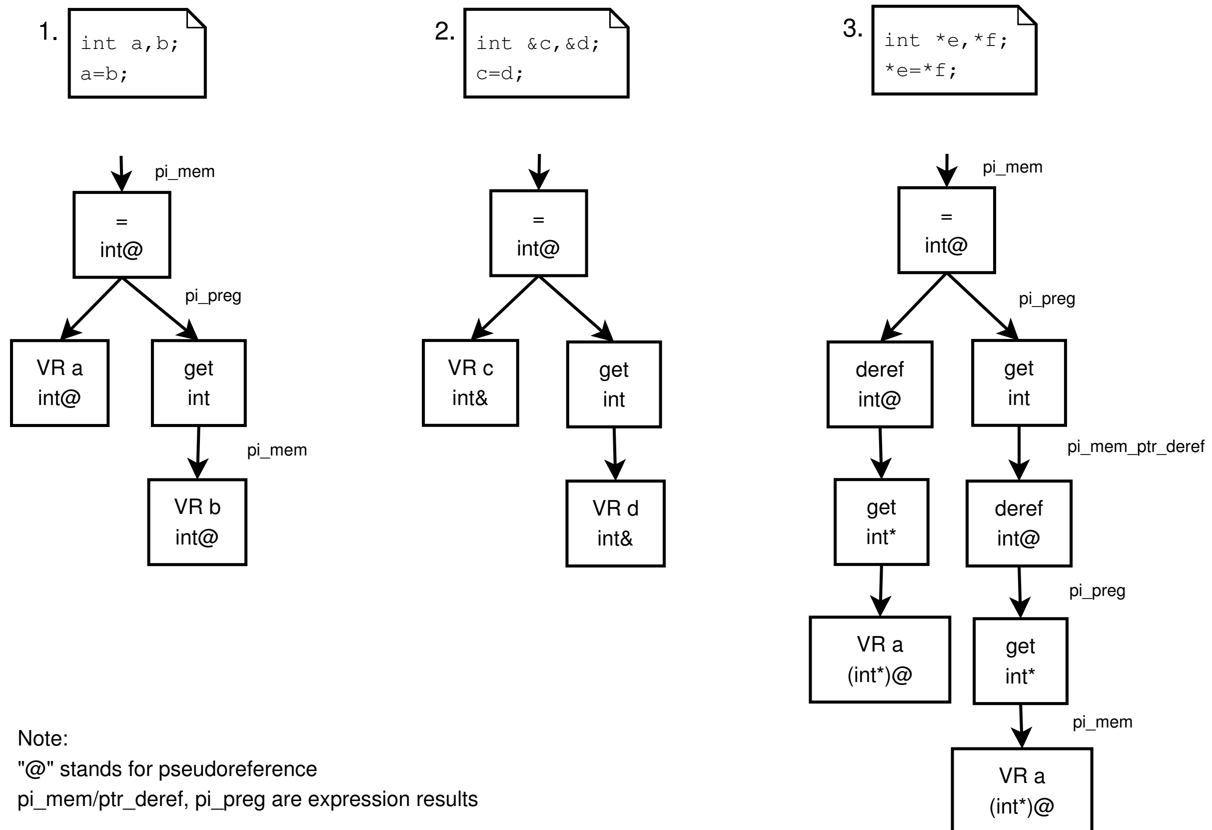
lvalues arise only in two situations—in *ss_var_ref*, which represents variables access (for details see 5.3.2) and *ss_deref*, which represents pointer dereference.

Next, lvalues can be used as input operands only in three cases:

- left operator in *ss_assign*, which represents assignment operator (=)
- in *ss_(vol)get*, which represents lvalue-to-rvalue conversion
- in *ss_bind_reference* which represents reference binding

Now we take example of *ss* structures in the picture.

[5]Namely, these operators are: `+, -, * (mul), /& (and), |, ^`, and with minor differences also `<<, >>`.



If we look at 1. and 3. diagram, we see, that in cases of get and assign, which have lvalues as operand, we have to discriminate between standard lvalue and dereferenced pointer to distinguish standard load/store instructions and load/store via pointer.

For these purposes we introduce method *ss_expr2pi.is_returned_pointer*, which will be called on given expression to make the distinction described above (it also evaluates that expression).

Next we take 2. and 3. diagram to demonstrate, how variable-reference in 2. diagram will be interpreted. Internally, we will consider such a reference as an ordinary pointer, and the only thing, which has to be changed, is *ss_var_ref* transformation—out of standard steps, *ss_get* and *ss_dereference* will be done, so transformation steps for 2. and 3. diagrams will be identical.

Details will be described in the following sections.

11.5.3 lvalue to rvalue conversion

This conversion is in the semantic structures represented as *ss_get* expression. Note, that in common case, when the variable is used in expression, only its value is used, which means lvalue to rvalue conversion is done. In backend intercode, such a conversion is reflected by simple load (*pi_ld*) from memory placeholder to a pseudoregister, which is usually the input operand for consequent pseudoinstructions. The second possibility is indirect load via pointer (*pi_ldp*) in case of conversion from dereferenced expression.

- Details: To distinguish these two cases *is_returned_pointer* is called first. This function evaluates expressions and it returns true when the expression is dereferenced. Operand with result is now just stored in *expr2operand_map* and it will be picked up, by standard transformation visitor call inside proper direct/indirect branch of pseudoinstruction construction.
- volatile lvalue to rvalue conversion is identical, only *ld(p)v* pseudoinstruction has to be chosen

11.5.4 Dereference

This semantic structure has no direct equivalent in the pseudoinstruction set—it just serves as a flag, that expression 'above' has to load/store its operand indirectly via pointer. It also encapsulate given pointer into *pi_mem_ptr_deref* type, to mark that given dereferenced pointer represents memory placeholder.

- Details: This flag is *ss_expr2pi.returned_pointer* attribute used by *returned_pointer* method.

11.5.5 var-ref

To prevent from confusion we shall distinguish in this section two things:

- variable-reference is variable of the type *type&*
- var-ref (represented by *ss_var_ref*) is *ss* structure referencing to usual variable used inside the expression.

In common case, the transformation consists of looking up the proper memory-placeholder for a given declaration of a variable. This memory-placeholder was previously allocated and initialized, when transforming declaration statement, declaring current variable.

The only thing which has to be handled occurs when the declared variable is a variable-reference. In such case lvalue-to-rvalue conversion of pointer representing the variable-reference is done and the flag for dereference is set. From this point on, variable-reference and an ordinary pointer are indistinguishable.

11.5.6 Assignment operator

An assignment in *ss* is represented by *ss_assign* expression and it is transformed into *pi_st* or *pi_stp* respectively in case of storing into dereferenced expression. Distinction between these two cases is solved just as in lvalue-to-rvalue conversion, see 11.5.3 .

- Details: in future development there could be pi_sti instead of pi_ldi & pi_st. Volatile store is not implemented yet.

11.5.7 Function calls

Function call transformation is more difficult than other expression. It consists of three parts and all of them have to be converted into equivalent pi structures.

11.5.7.1 Parameters of funcall

Each parameter, which is *ss_expression*, is firstly converted by standard visitor. Result in *pi_operand* is then passed to backend copy constructor (*copy_constructor_call_generator*) which copies argument-value to a temporary variable on the stack (created by *function_parameter_allocator*), which will be consequently used inside the called function.

11.5.7.2 Returning value

Returning value has to be especially handled by backend allocator *mem_alloc_manager::allocate_fcall_par_ret_val* , because it can be returned in many different ways according to the target architecture.

11.5.7.3 Type of funcall itself

There are two orthogonal distinctions of funcalls in *ss*:

- a) Calling a function
  - 1. specified by name (*funcall*, *mfuncall*, *vfuncall*)
  - 2. nominated by an expression (*ifuncall*, *pfuncall*)
- b) Calling
  - 1. member function (method)
    - i. with static binding (*mfuncall*, *pfuncall*)
    - ii. with dynamic binding through VMT (*vfuncall*, *pfuncall*)
  - 2. non-member function (namespace function) (*funcall*, *ifuncall*)

There are two orthogonal distinctions of funcalls in *pi*:

- c) Calling a function

  - 1. indirectly through pointer (*icall*,*icallv*)
  - 2. directly (*call*,*callv*)

- d) Calling a function
  - 1. with return value (call, icall)
  - 2. with no return value (void) (callv, icallv)

The transformation can be described as a map:

ss	->	pi
a1 b1i	->	c2 d1/2
a1 b1ii	->	c1 d1/2
a1 b2	->	c2 d1/2
a2 b1i	->	c1 d1/2
a2 d1ii	->	c1 d1/2
a2 b2	->	c1 d1/2

11.5.7.4 Transformation details

Funcall transformation can be abstractly written as:

- 1. *pi_call* construction.
- 2. Return value and *this* (secret) argument allocation, if any is needed (note: temporary for return value shall be generated such that its origin is the affected funcall.) Destructor table is updated for the return value.
- 3. Transformation of arguments. For each argument:
  - i. Recursive argument expression transformation to pi.
  - ii. Allocation of temporary variable for the argument, which will be used inside the function through *function_parameters_accessor*.
  - iii. Construction of copy constructor of parameter (also for simple types). Note: this is the right place for copy-constructor elimination.
  - iv. Adding proper destructor into *temporaries* destruction table.
- 4. Determining pseudoinstruction level (1 + maximum from sequence points and funcall arguments).
- 5. Passing pseudoinstruction call to backend.

11.5.8 Comparison operators

This group of operators[6] are now directly transformed into corresponding pseudoinstructions.

There is possible enhancement in connection with *if statement*. Up to now, comparison is being done and its result is stored as boolean value. Then the branching instruction jumps accordingly to previous result. This can be improved via direct using of compare-branching instructions[7].

11.6 Literals

For literal transformation, two steps are done. Firstly literal data(*ss_literal_info*) are transformed to backend literal quality and then they are loaded into proper pseudoregister. Since the target-machine type representing the literal can affect the kind of loading pseudoinstructions, this part is done by backend *literal_loader*.

11.6.0.1 Literal_info transformation

Transformation of semantic literal quality *(ss_literal_info)* to backend literal quality (*pi_literal_info*) is done via *ss_linfo2pi_linfo* visitor. Structurally are these two qualities identical, and standard *ss_type_to_tm_type_convertor* is used for internal data conversions.

[6]Namely sbg(>), sbl(<), sbe(==), sbng(<=), sbnl(>=), sbne(!=)

[7]Namely bg, bl, be, bng, bnl, bne

11.7 Used backend interface in ss2pi

11.7.0.2 Passing pi structures to backend

This is designed according to builder design pattern. Assembled pseudoinstruction and its operands are passed to *backend_data_builder.add/add_sp()*, which is responsible for enqueing into backend structures. After complete transformation of particular ss structure, builder is informed by *backend_data_builder.rail()* function, which can be useful for debugging.

11.7.0.3 Conversion of types to backend

Type conversion cant be maintained by semantic part itself, because target architecture information is required and backend *ss_type_to_tm_type_convertor* API is used.

11.7.0.4 Memory management

Allocating memory-place-holders are done using backend *mem_alloc_manager* API. Pseudoregisters do not have to be allocated via a specialized backend interface and are created directly in ss2pi code.

Some data manipulations - such as return value passing, copying of temporaries on the stack, or function parameters passing cannot be done directly, because its target-dependent which actions are to be done. For this task backend *copy_constructor_call_generator* API is used. It returns proper pseudoinstructions to ss2pi and finally are these pseudoinstructions passed back via *backend_data_builder* API to backend.

11.8 ss2pi transformation API

ss2pi transformation provides only one API function *ss2pi_start* for the rest of compiler. Its argument is a translation unit, which is to be transformed and passed to backend.

11.9 To be implemented summary

This is summary of the features in the ss2pi module, which should be implemented in a future work

- class-related code (ss_de/con/structor_finders, VMT, accessing operators, related funcalls)
- namespaces
- exceptions
- some expression operators: ||, &&, ?:, ifuncall, pfuncall
- some statement operators: break, continue, goto, switch, do, for

11.10 Transformations summary

11.10.1 ss layer to pi layer

ss layer	**pi layer**
Expressions	
ss_assign	pi_st/stp
ss_var_ref	get_mem/(pi_ld+ptr_deref)
ss_(vol)get	pi_ld(v)/ldp(v)
ss_literal	pi_ldi
ss_dereference	ptr_deref
ss_address_of	pi_lda
ss_bind_ref	pi_lda
ss_funcall	pi_call/calv
ss_add/sub/mul/div/mod	pi_add/sub/mul/div/mod
ss_shr/shl	pi_shr/shl
ss_sbg/sbl/sbe	pi_sbg/sbl/sbe
ss_sbng/sbnl/sbne	pi_sbng/sbnl/sbne
ss_gat/neg/gat	pi_get/neg/gat
ss_conversion	pi_cvt
Unimplemented	
ss_land/ss_lor	pi_bf/pi_bt
ss_ternary	pi_bf+pi_bt
ss_x-funcall	pi_call*[8]

ss_layer	**pi_layer**
Statements	
ss_if_stmt	pi_bf+pi_ba
while	pi_bf+pi_ba
ss_return	copy_constructor+pi_leave
Unimplemeted	
ss_switch	pi_bm
ss_do/for	pi_bf+pi_ba
ss_continue/break	pi_ba

11.10.2 Exception transformation

Exception statement	**Transformation**[9]
catch (handler) try block	decl_stmt + compound_stmt compound

Chapter 12

Code Generation

12.1 Introduction

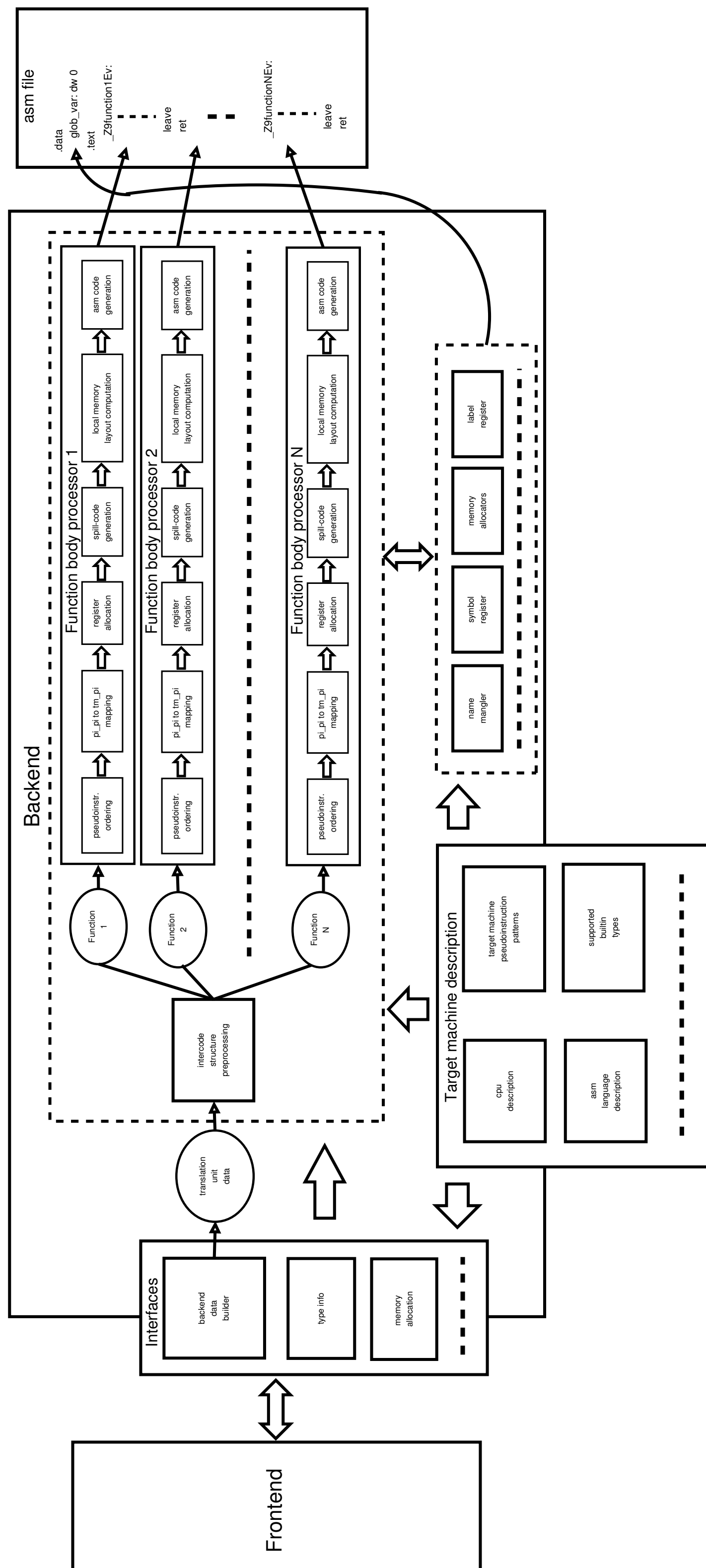
The goal of code generation module (backend) is to transform intercode structures passed from frontend to output assembly source. Backend is as simple as possible. Basic demand is ability to generate assembly code for simple C++ source. Therefore no optimizations are implemented and the generated code is by no means effective.

The code generator uses standard C++ name mangler and function calling sequence, so the output asm code can be compiled by external assembly compiler and its output object file can be linked with standard C++ library into binary executable.

12.2 Directory Structure

- The backend's target-independent sources are located in `/lestes/backend` subdirectory of the lestes compiler. There are following directories:
  - `debug` - debugging stuff.
  - `intercode` - pi-level intercode classes and visitors.
  - `interface` - interfaces that backend provides to other modules.
  - `modules` - function modules.
  - `structs` - helper classes.
  - `target` - classes that describes target-machine.
- Target-dependent sources are located in `/target/machine/${cpu}/lestes/backend/ (${cpu}` is replaced by cpu identification of architecture for which output code is generated). Structure of this directory is similar with independent one.
- XML description of specific architecture is can be found in `/target/machine/${cpu}/tm_description/machine_descrip` file.
- XSD schema for machine description document validation is located in `/util/machine_description/schema/target_arc`

12.3 Backend schema



12.4 Machine Description

Base of backend is description of target architecture - machine description (MD). The MD contains relevant information on the architecture and its basic software environment, namely:

- the list of hardware resources of the architecture (registers)
- supported data types

- description of instruction set
- outline of assembler syntax and its basic lexical elements

These information are required for register allocation, instruction selection, asm code generation, ...

The machine description system is divided into three parts:

12.4.1 XML Description File

MD file language is based on XML. This language is used to describe hardware resources of the architecture, instruction set, asm syntax and rules for instruction selection.

Elements:

12.4.1.1 DataType

Describes data type supported by architecture.

```
<DataType id="DT_INT_32S" format="C2" bitwidth="32" alignment="32" mapped_type="ss_type_sint"
return_register="R_EAX" asm="DWORD" asm_decl="dd"/>
```

Attributes:

- `id` - id of type. It is used by other elements to reference the type.
- `format` - data type format (plain, 1-complement, 2-complement, BCD, IEEE,....).
- `bitwidth` - width of the type in bits.
- `alignment` - alignment of the type in bits.
- `mapped_type` - ss-level datatype that is mapped to this target type.
- `return_register` - register where return value of the type is returned from function.
- `asm` - typename used in assembler output.
- `asm_decl` - typename used in assembler output to specify type in variable declaration.

12.4.1.2 Register

Describes a register of target machine processor.

```
<Register id="R_EAX" bitwidth="32" compatible_types="DTG_INT_32" aliases="R_AH R_AL R_AX"
asm="EAX" />
```

Attributes:

- `id` - id of register. It is used by other elements to reference the register.
- `bitwidth` - width of the register in bits
- `compatible_types` - data types of object that can be stored in the register.
- `aliases` - registers that are aliases for the register. Simultaneous usage of these registers is not possible. (e.g. i386 has register AX that is 16-bit wide alias for 32-bit register EAX). This information is important for register allocator.

12.4.1.3 PseudoInstruction

Describes `tm_pi` pseudoinstruction supported by target processor.

```
<PseudoInstruction id="PI_MUL_1" pi="mul" />
   <InOperands>
      <Register id="I1" datatypes="DT_INT_32S" allowed_registers="R_EAX R_EBX R_ECX R_EDX"/>
   <OutOperands>
      <Register id="O1" datatypes="DT_INT_32S" allowed_registers="R_EAX"/>
   </OutOperands>
   <Requirements>
      <Registers ids="R_EDX" />
   </Requirements>
   <Output>
      <Asm>mov $O1,$I1</Asm>
      <Asm>imul $I2</Asm>
   </Output>
</PseudoInstruction>
```

This example defines 32bit signed multiplication.

Atributes:

- `id` - id of tm_pi instance.
- `pi` - pi-level pseudoinstruction `pi_pi` that is mapped to this `tm_pi`. NOTE: more than one possible `tm_pi` can exists for single `pi_pi`. The right one is found by operand matching in instruction selection module12.5.2.2.

Sub-elements:

- `InOperands` - contains list of input operands.
- `OutOperands` - contains list of output operands.
- `Requirements` - resources required for execution of the pseudoinstruction.
- `Output` - output asm template.

12.4.1.4 Pseudoinstruction Operand

There is several operand types:

1. `<Register id="I1" datatypes="DT_INT_32S" allowed_registers="R_EAX R_EBX R_ECX R_EDX"/>` - represents operand that is placed in register.
2. `<Memory id="O1" datatypes="DT_INT_32S DT_INT_16S DT_INT_8S" />` - represents operand that is placed in memory.
3. `<Literal id="I2" datatypes="DT_INT_32U" />` - represents literal (constant).

Attributes:

- `id` - id of operand within pseudoinstruction. String in the output asm template that equals the id is to be replaced by asm string that represent real operand during asm code generation phase of the backend (e.g. in case of register I1, $I1 is replaced by EAX in output template).
- `datatypes` - allowed data types for the operand.
- `allowed_register` (valid within `Register` element only) - set of registers where the operand can be placed.

12.4.1.5 Groups, Templates

In order to simplify notation of `PseudoInstruction`, there are introduced following elements:

- `<DataTypeGroup id="DTG_INT_32" types="DT_INT_32U DT_INT_32S DT_LONG_32U DT_LONG_32S" />` - defines group of datatypes. Its id could be used in attributes where set of datatype ids is allowed. It expands to type ids.
- `<RegisterGroup id="RG_GP_32" bitwidth="32" registers="R_EAX R_EBX R_ECX R_EDX R_EDI R_ESI" compatible_types="DTG_INT_32" />` - defines group of registers. Its id could be used in attributes where set of register ids is allowed. It expands to register ids. Attribute `compatible_types` defines set of datatypes that are compatible with all registers in the group.
- `PseudoInstructionTemplate` - allows to define template for operands. It could be used by multiple `PseudoInstruction` to define its operands.

```
<PseudoInstructionTemplate id="PI_TEMPLATE_ARIT1">
   <InOperands>
      <Register id="I1" datatypes="DTG_INT" allowed_registers="RG_GP_8 RG_GP_16 RG_GP_32"/>
   <OutOperands>
      <Register id="O1" datatypes="DTG_INT" allowed_registers="RG_GP_8 RG_GP_16 RG_GP_32"/>
</PseudoInstructionTemplate>
```

- `OutputTemplate` - allows to define template for output. It could be used by multiple `PseudoInstruction` to define its `Output`.

```
<OutputTemplate id="O_TEMPLATE_DIV2">
   <Output>
      <Asm>mov AX,$I1</Asm>
      <Asm>cwd</Asm>
      <Asm>idiv $I2</Asm>
   </Output>
</OutputTemplate>
```

  In order to use templates within `PseudoInstruction`, attributes `pi_template` and `out_template` are introduced.

  Example:

```
<PseudoInstruction id="PI_DIV_1" pi="div" pi_template="PI_TEMPLATE_DIV1"
out_template="O_TEMPLATE_DIV1"/>
```

12.4.1.6 Assembly Language Elements

```
<Assembler>
   <Entities>
      <Entity id="ent_address_deref">[$address]</Entity>
      <Entity id="ent_section_data">\nsection .data\n</Entity>
       ....
   </Entities>
</Assembler>
```

Each `Entity` element defines asm language element. Output asm file is assembled of these elements.

12.4.2 Generated C++ Code

Parsing of MD file and mining of required data in runtime of compiler would slow down entire compilation. Therefore information about target architecture are compiled into compiler binary.

There are defined C++ classes that are equivalent to the MD's elements:

- `tm_data_type`
- `tm_register`

- tm_pi
- tm_operand

Before compilation of compiler, the MD file is processed by multiple XSLT that generates C++ sources with implementation of instance() methods for these classes. The instance() method returns instance initialised with data of object with the same id from MD file .

Tables for target pseudoinstruction selection 12.5.2.2 are also generated from MD file. For each pi_pi subclass there is generated list of ids of possible tm_pis. One of them is chosen during pseudoinstruction selection.

12.4.3 Hand-Written Classes

Some information about target architecture are too complex to be simply described in MD file. Therefore the information are represented by hand-written classes. They have to be implemented for every target architecture that the lestes supports. They are placed in target-dependent directory12.2. There are mandatory interface names and methods for each class that an implementation has to implement.

12.4.3.1 Memory Allocators

Memory allocator allows to allocate memory. Allocated memory chunk is represented by pi_mem_factory 6.3.1 class. It also works as factory for pi_mem6.2.1 objects.

- local_variable_allocator is manager for local memory allocation. It allocates/deallocates memory chunks on function's local stack. Base class of the interface is **local_memory_allocator_base**.
- global_variable_allocator is manager for global memory allocation. It allocates/deallocates memory chunks in global data section. Base class of the interface is **global_memory_allocator_base**.
- funcall_parameter_allocator allows to allocate space for parameters passed to function calls. The space is allocated on caller function stack. Base class of the interface is **funcall_parameter_allocator_base**.

Placement of variables of some types can't be determined at the time of allocation (e.g. placement of local variable can't be done until total count of local variables is known). So every type of memory allocator has method calculate_pacement() that calculates placement of all objects that have been allocated by the allocator. The calculate_placement() method should be called at the point when no more new memory is to be allocated.

12.4.3.2 Function Parameter Access

It allows to access parameters of a function within the function. Parameters of the function are placed on the caller stack. The function_parameter_accessor returns pi_mem_factory6.3.1 object that represents the space. Base class of the interface is **function_parameter_accessor_base** class.

12.4.3.3 Symbol register

The symbol_register holds list of global symbols (functions) defined within translation unit. The symbol_register is used to generate global asm symbol declarations for all the registered symbols during asm code generation phase. Base class of the interface is **symbol_register_base** class.

12.4.3.4 Data type ranges

The num_range_getter provides additional information about numeric data types such as maximum value, maximum digit count, range of a exponent and range of a mantissa. Base class of the interface is **num_range_getter_base** class.

12.4.3.5 Literal Information

The pi_literal_info6.3.2 class is target dependent because it has to know how to emit asm representation of the literal it represents. Base class of the interface is **pi_literal_info_base** class.

12.4.3.6 Operand Output Code Generator

Although the MD file contains information about basic asm language elements, it is not easy to put these pieces together to create asm code for pseudoinstruction operand. The visitor visitor_pi_operand2asm implements code emitor for operands. Base class of the interface is **visitor_pi_operand2asm_base** class.

12.4.3.7 Pseudoinstruction Output Code Generator

The visitor_pi_pi2asm generates asm code for given pseudoinstruction in the following steps:

- It gets output asm template from tm_pi that the pseudoinstruction is mapped to.
- For each operand of pseudoinstruction it replaces operand's tag with operand code12.4.3.6 (with register name in case of pseudoregister, with literal value in case of literal and with address in case of memory operand).
- If the pseudoinstruction is a branch6.1.5 then it replaces tag for jump target with label name.

This class is added among target dependent classes because of specifics that a target architecture could have. Base class of the interface is **visitor_pi_pi2asm_base** class.

12.5 Backend Processes

After the frontend has processed C++ source file (translation unit - TU) it creates pi-level intercode6 structures that are passed to backend for further processing. This section describes transformations of these structures that are needed to output file in assembly language done by backend module.

12.5.1 Passing input intercode structures to the backend

Input intercode structures for TU are passed to the backend in class **backend_data_builder**. The following functions are used to feed the data into the builder object. It builds internal data structure backend_data which is suitable for backend.

- void backend_data_builder::add_function_start(ptr<ss_function_declaration> function) - lets the builder know, that it is supposed to start building structure of another function body.
- void backend_data_builder::add_sp(ptr<pi_sp> sp) - adds pi_sp to the currently built body.
- void backend_data_builder::add_pi(ptr<pi_pi> pi) - adds pi_pi to the currently built body.
- void backend_data_builder::add_pis(ptr<list<srp<pi_pi> > > pis) - adds list of pi_pi to the currently built body.
- void backend_data_builder::add_rail() - lets the builder know that processing of particular semantic structure has been finished.
- void backend_data_builder::add_function_end() - lets the builder know, that all pseudoinstructions composing body of the currently built function have been passed.

12.5.2 TU processing

Main class of the backend is called **backend.** Backend takes input data from backend_data_builder and preprocesses it. List of global symbols (function declarations) and list of labels (sequencepoints) are collected during preprocess phase. After that, backend generates asm declarations of global symbols to the output asm file. TU is composed of function bodies. Processing of each function can be done with minimal interaction with the rest of TU. So backend splits input structures to blocks (functions) and passes them one by one to block_processor for further processing.

The block_processor takes single body and transforms it to asm code. The code is then appended to the output asm file. The transformation is done by modules and purpose of the block_processor is to forward temporary results among them.

Implemented block_processor modules:

12.5.2.1 Pseudoinstruction ordering

Pseudoinstructions are not passed to `backend_data_builder` in order in which they will be executed. So for a couple of pseudoinstructions from input body one could not decide which pseudsoinstruction will be executed first. **`Order_governor`** module orders them with iterative topological ordering algorithm. Result of the `order_governor` is ordered body where each pseudoinstruction is placed between its boundary sequencepoints and after origins of its operands6.2.

Definitions:

- **Psp-nsp** graph is a pair consisting of sets V and E. The set V is a arbitrary set called set of vertices. The second set $E \subseteq \{V \times V \times \{n, p\}\}$.
- We say the triple $< u, v, n >$is a **nsp edge** from u to v. Similarly the triple $< u, v, p >$is called a **psp edge** from u to v. For each but one vertex of psp-nsp graph there exists exactly one psp edge leading to it. One of the vertices has no psp edge leading to it. Similarly for each but one vertex there exists exactly one **nsp edge** leading from it. One of the vertices has no nsp edge leading from it.
- **Conjugated pair** of vertices is pair of vertices $< u, v >$where there is a psp edge leading from u to v and there is simultaneously a nsp edge leading from v to u. We say that u is the **start of the conjugated pair** and that v is the **end of the conjugated pair**.
- We say that a sequence of vertices forms a **conjugated chain** if each two adjacent vertices form a conjugated pair. The first vertex in the sequence is called the **start of the conjugated chain**. The last vertex is called **the end of the conjugated chain**.
- We say that a **conjugated chain is maximal** if there is no vertex that forms a conjugated pair with start of the chain nor there is vertex that forms a conjugated vertex with end of the chain. A degenerated sequence of single vertex is also considered conjugated chain.
- We say that two vertices are connected by a **psp path** when there exists a path composed of only psp edges. Similarly we say that two vertices are connected by **nsp path** when they are connected by path composed only by nsp vertices.
- We say that u is **ordered less** that v when there exists a psp path from u to v or a nsp path from u to v.
- We define the **psp** of a vertex u to be the single vertex from which there exists a psp edge leading to u. Similarly the **nsp** of vertex u is defined as the only vertex to which leads a nsp edge from u. If there are no such vertices that the vertex u has no psp or nsp.
- The psp-nsp graph carry the property that the only vertex that has no psp and the only vertex that has no nsp are the members of the same conjugated chain. For each starting vertex u of a maximal conjugated chain and for the ending vertex v of the same chain (this can be the same vertex in the case of degenerated conjugated chain) the psp of u and shall be ordered less than nsp of v. These requirements are indivisible part of the psp-nsp graph definition.
- An non-negative integer called **level** is assigned to each vertice in psp-nsp graph. We define the level of vertex recursively as follows:
  - All vertices of a single conjugated chain have the same level.
  - The level of a vertex that is the start of maximal conjugated pair is strictly greater than the level of its psp.
  - The level of the end of a maximal conjugated chain is strictly greater than the level of its nsp.
  - The level of non-existent psp or nsp is defined to be minus one.
  - The level of any vertex is the smallest integer satisfying all the preceding conditions.
- We define $l_i$to be the set of vertices with the level equal to i. We call the l_i the ith**layer.**
- We define L_i to be psp-nsp graph that is composed of vertices from $\cup_{j=0}^{i} l_j$.
- We define $H_i = \cup_{j=i}^{\infty} l_j$.

Iterative Topological Ordering (ITO)

The ITO algorithm is designed to perform the ordering of pseudoinstructions. It is expressed in therms of transformation of a psp-nsp graph.

1. A topological ordering is performed on L_1.
2. All the vertices from $H_2$are examined and their psps and nsps are moved.

   The moving of the psps and the nsps is done as follows:

   If the psp of the vertex v is member of L_0then the psp of v is changed by following - nsp path leading of the former psp to the point where it reaches either a vertex with level greater than one or the vertex that is the psp of the end of the maximal conjugated chain the vertex being considered is a member of. A similar process is then made for adjusting the nsp with the roles of psp and nsp, and start and end of conjugated chain reversed.
3. The topological ordering merges the layers $l_0$and l_1, therefore all the levels are recalculated.
4. The process starts over, unless there are no vertices with level greater than zero.

Note:

- psp is previous sequencepoint of a pseudoinstruction.
- nsp is next sequencepoint of a pseudoinstruction.

12.5.2.2 Target pseudoinstruction selection

The **`pi2tm_mapper`** module chooses `tm_pi` pseudoinstruction for each `pi_pi` pseudoinstruction `pi` of the body. First it gets list of ids of possible `tm_pi` instances for `pi` from machine description 12.4.2. Then it creates instance `tpi` of `tm_pi` for each id from list and checks whether the mapping exists. The mapping exists if the following conditions are satisfied:

- count of `pi`'s input operands is equal to count of `tpi`'s input operands.
- kind (memory,register,literal) of nth `pi`'s input operand is the same as kind of nth `tpi`'s input operand.
- data type of nth `pi`'s input operand is the same as data type of nth`tpi`'s input operand.
- similar conditions have to be fulfiled for output operands.

12.5.2.3 Register allocation

A register allocator tries to map pseudoregisters to target-architecture registers.The **`register_alocator`** module performs dummy register allocation. It goes through function body and for every pseudoregister operand sets flag that suitable free register is not found. These unsuccessfully allocated pseudoregisters are handled by spill_code_generator module.

12.5.2.4 Spill-code generation

Sometimes register allocator runs out of free register because of fixed number of target cpu's registers. The **`spill_code_generator`** module goes through function body and checks pseudoregisters whether each of them has its register assigned. If it finds a pseudoregister without assigned corresponding register then it generates spill-code. Spill-code is generated in order to temporarily free any of allocated registers and assigns it to the pseudoregister. Spill-code generation for pseudoregister `P` within pseudoinstruction `PI` has several steps:

1. It gets memory space `M` that is assigned to `P` (memory space is assigned by `spill_code_generator` to every pseudoregister without register at the first pseudoinstruction where it is used). `M` holds value of `P` between consecutive uses.
2. It chooses register `R` which can hold datatype of `P`, can be used in `PI` and is allocated to pseudoregister that is not used within `PI`.
3. It generates `store` pseudoinstruction in front of `PI` that moves value of `R` to a temporal memory space `T`.

4. It temporarily assigns R to P within PI.
5. If P is input operand of PI then it generates load pseudoinstruction in front of PI that loads value of M to R.
6. If P is output operand of PI then it generates store pseudoinstruction behind PI that stores value of R to M.
7. It generates load pseudoinstruction behind PI that loads original value of R from T back to R.

Because of dummy register allocator, the choosen register from step 2 is empty and therefore preserving its value can be skipped (steps 4 and 7) .

12.5.2.5 Local Memory Layout Computation

Before the final code-generation stage is started, there have to be computed layout of function's local stack (local variables placement). This work is done by memory allocators 12.4.3.1.

12.5.2.6 Assembly code generation

The **asm_generator** module passes through the function's body and generates12.4.3.7 assembly code for each pseudoinstruction.

12.6 Interfaces

Backend defines interface that can be used by other modules of the Lestes compiler to obtain information about target architecture or to utilize services provided by backend. The interface provides the following services (description of interface methods can be found in /doc/backend_interfaces.txt):

12.6.1 Memory allocators

The backend provides interface **mem_alloc_manager** for local and global variables allocation. The allocation means that somebody lets the backend know about a new variable, the backend returns object representing a corresponding place in memory and at the end it generates assembler definition for this object. This interface is wrapper over backend's internal memory allocators12.4.3.1.

12.6.2 Function's parameter access

See 12.4.3.2

12.6.3 Literal loading

The interface **literal_loader** generates list of pseudoinstructions that load value of a literal to a register. Generated pseudoinstructions differ according to type of literal. If literal can act as immediate then it is loaded by single pi_ldi pseudoinstruction. Global asm declaration is generated otherwise (managed literal) and literal is referenced through its address.

12.6.4 Copy-constructor call generation

The **copy_constructor_call_generator** generates list of pseudoinstructions that copy value of pi_operand operand to pi_mem operand. Generated list depends on:

- If data type of operands is a builtin or POD type (copy-constructor not defined), then the function creates list of pseudoinstructions that copies value of the input operand into the output operand. Otherwise function creates list containing single pi_function_call pseudoinstruction which represents copy constructor call for the type.
- kind of source operand - reg -> mem (single store pseudoinstruction) vs. mem -> mem (load source to temporary register and then store temporary to destination memory)

12.6.5 Type information

The **type_info** provides information about data types such as ranges, information about mantissa, exponent, etc.

12.6.6 Name mangling

The **name_mangler** performs C++ name mangling (compatible withGNU gcc mangling `http://www.codesourcery.com/cxx-abi/abi.html#mangling`). It gets `ss_declaration` object and returns mangled name. If the given `ss_declaration` is not C++ declaration (e.g. external C function declaration) then it returns just the name of the declared object.

12.7 Debugging

Backend has the following support for debugging:

- internal data-structure dumping - if there is the symbol `BACKEND_DEBUG` defined, then the dumps 14.11 of internal structure of processed function body are created after each block_processor module.
- loggers - there are defined multiple loggers18 that log events during translation unit processing.

12.8 Things to Be Implemented

- Classes/Structures - support for classes and structures is not implemented although there are defined subclasses of `tm_data_type` for structured data type. To implement this feature it is necessary to implement the following:
  - data types for member-data pointer and member-function pointer
  - member functions
  - virtual functions & virtual method table (VMT)
  - bitfields
  - name mangler extension that mangles class and namespace declarations.
- Globally initialized non-POD data - a global variable of basic data type (POD, with no copy-constructor defined) can be now initialised with literal(constant). Initialization of non-POD data type requires generating copy-constructor call to a special global initialization function. More informations can be found in C++ ABI `http://www.codesourcery.com/cxx-abi/abi.html`.
- Unsupported pseudoinstruction emulation - it can happen (on some architectures) that for some `pi_pi` pseudoinstruction there is no compatible `tm_pi` pseudoinstruction. Emulation of such a pseudoinstruction requires extension of the machine description with a library of functions emulating the pseudoinstructions. Then in the asm generation stage there is generated function call of corresponding emulating function instead of instructions for the emulated pseudoinstruction.
- Optimizations
  - register allocation
  - instruction scheduling
  - ...
- Support for exported templates
- Other stuff mentioned in C++ ABI `http://www.codesourcery.com/cxx-abi/abi.html` such as:
  - array operator new cookies
  - guard variables
  - support for RTTI

- array construction and deconstruction API
- controlling object construction order
- DSO object destruction API
- exceptions

Part IV

Framework

Chapter 13

Directory Structure

This section gives the basic overview of the directory structure of the project tree, explains contents of individual directories to ease the orientation and navigation through the project tree.

- `build` - this directory contains the main Makefile for the whole project.
- `host` - this directory contains the files which are target-architecture dependent. Currently this includes definition on host types for i686.
- `lestes` - this directory contains the source codes of the architecture-independent part of the compiler.
  - `backend` - the target-independent part of the backend.
    - `debug` - debugging stuff.
    - `intercode` - pi-level intercode classes and visitors.
    - `interface` - interfaces that backend provides to other modules.
    - `modules` - function modules.
    - `structs` - helper classes.
    - `target` - classes that describes target-machine.
  - `intercode` - this directory contains the abstract class for the intercode representation
  - `lang` - this directory is intended for source-language specific sources. Currently, as the lestes compiler is supporting only C++, there is located the only directory
    - `cplus`
      - `lex` - this directory contains implementation of C++ lexical analyzer and preprocessor
      - `sem` - this directory contains implementation of C++ semantic analysis
      - `syn` - this directory contains implementation of C++ syntactic analysis (i.e. parser)
  - `msg` - this directory contains the implementation of error reporting framework and also implementation of logger
  - `std` - this directory contains the implementation of wrappers for STL classes for purposes of garbage collecting, and implementation of lestes smart pointers.
    - `mem` - this directory contains the implementation of lestes memory management and garbage collector
- `lib` - this directory contains the implementation of the library used by our compiler for demonstration purposes
- `runner` - this directory contains, after a successful compilation, the main lestes binary
  - `examples` - this directory contains the example source codes, which demonstrate individual features supported by lestes compiler
    - `fail` - this directory contains the example source codes, which demonstrate that lestes compiler correctly detects source codes which are malformed in some way

- `target` - this directory contains target-dependent parts of backend of the lestes compiler for individual supported architectures. The structure of this directory resembles the structure of the directory containing the target-independent part.
- `util` - this directory contains various utilities used to compile the project tree. This includes XSL templates for structure generator, helper to examining ELF object files and give information what object files are required when linking, XSL templates for transformation of intercode structures to HTML, etc.

Chapter 14

Structure Generator

The goal of the generator is to allow easy maintenance of complex data structures used at different compilation levels. It embraces hundreds of classes with some common parts, which can be generated automatically. Support for garbage collection, dumping, visitors etc. becomes nearly transparent this way.

The generator is written in the XSL Transformations language `http://www.w3.org/TR/xslt` (XSLT) and takes XML `http://www.w3.org/XML/` files as input. Result is as C++ code.

14.1 Data Formats and Processing

There are two types the of source XML files: Lestes Structures Description (LSD, stored in *.lsd files) and Lestes Visitors Description (LVD in *.lvd files). LSD is used to describe several classes, their members, inheritance and special properties. LVD is used by the visitors-generating mechanism, see the Visitors 14.14 section below.

To process LSD and LVD files a XSLT processor is needed, such as **xsltproc**. The process is fully supported by our build system 15.

There are several stylesheets (*.xslt) in `util/lsg` directory, but the most important are **lsd2hh**, **lsd2cc**, **lvd2lsd**, and **lvd2cc**. The lsd2* take an `X.lsd` file as input and create one `X.g.cc` and one `X.g.hh` file (g as "generated"). If there are some methods to be implemented by hand, it should be placed in hand-written `X.cc` file. lvd2* stylesheets take a LVD file to generate `Y.v.lsd` and `Y.v.cc` (v as "visitors"), see the Visitors 14.14 section below.

14.2 XML Validation

Validating a file as a correct LSD or LVD file is not possible at this time. A XML Schema `http://www.w3.org/XML/Schema` for LSD and LVD should be written ASAP. The generator itself does no sort of checking and if a bug appears in an input file, the generated code may be wrong or the XSLT processor may fall down.

14.3 Comments

Anywhere in LSD or LVD file a `<comment>` element may occur. Its content is ignored by generator. An alternative is a XML-like comment:

```
<!-- This text is ignored by XML parser. -->
```

14.4 LSD File Header

14.4.1 Root Element and XML Namespaces

It is recommended to watch `util/lsg/example.lsd` and other examples in the directory along with creating LSD files. The first two lines and root element name are common for all LSD files.

```
<?xml version="1.0" encoding="UTF-8"?>
<lsd xmlns="http://lestes.jikos.cz/schemas/lsd" xmlns:h="http://www.w3.org/TR/REC-html40">
```

```
...
</lsd>
```

The xmlns:h attribute can be omitted if we do not use HTML tags in Doxygen comments 14.9.

14.4.2 LSD Prologue

The header part of LSD contains the following elements in this order:

```
<dox/>
<file-name/>
<packages/>
<imports/>
<implementation-imports/>
<hh-header/>
<hh-footer/>
<cc-header/>
<cc-footer/>
<include/>
<default-base/>
<default-collection/>
<default-check/>
```

Any of these elements may be omitted or left empty if we don't need it. Only the <file-name> element is required and contains the name of the LSD file without extension. It must not contain any peculiar characters because it's used as a part of an identifier in output C++ code. Dashes and dots are allowed.

The <dox> part contains <bri> and <det> elements, which store a doxygen documentation for files generated from this LSD (brief and detailed). There may be two <dox> elements with file="hh" and file="cc" attributes, or only one with file="both". The attribute says whether the documentation describes only the header or implementation file or both. See also the Doxygen 14.9 section below.

The <imports> element contain any number of <i> elements, each of them containing a path to a file. Such <i> element corresponds to one #include line in output *.g.hh header file. The file name is always bounded by < and > brackets, because relative including with quotes is prohibited in Lestes project.

The <implementation-imports> part is analogous to the previous one, but the #include lines are generated only into *.g.cc implementation file instead of the header file.

The <packages> element is to satisfy the namespace policy in Lestes project. It contains package names in <p> elements. All classes generated from this file are placed into one namespace determined by these package names. See also the Napespaces section 14.13 below.

The following elements:

```
<hh-header/>
<hh-footer/>
<cc-header/>
<cc-footer/>
```

may be used to add a few hand-written code into the generated C++ files.

The <include> element may occur arbitrarily many times and has one required attribute - href. It contains a path to another LSD file. It's purpose isn't a true inclusion such as in C++. The only purpose of these links is to make other generated classes in the same namespace visible for generator, if they are bases of hereby classes. A generated constructor or factory method 14.12 must initialize all data members of current class, including the inherited ones. Therefore parents must be accessible for generator. Note that the search for bases is done recursively, so you should include only the LSD file with the nearest descendant. The cycling in inclusion is not treated, so users must avoid it. And it is also bad if some class can be reached more than one way via includes. However it is easy to obey these restrictions if the <include> hierarchy is a tree corresponding to the natural ISA hierarchy between classes.

The <default-base> element has a type attribute. It denotes a base class for all classes with the base attribute omitted.

The <default-collection> element has a kind attribute. It denotes a kind for all collection members with kind attribute omitted.

The <default-check> element may contain a name of a macro to be used as default for this LSD file instead of checked(arg). Function of this macro is described below.

Some helper declarations follow the header part: <forward-class>, <using-class>, and <foreign-class>. Their usage is described below in sections Garbage collection 14.10, Namespaces 14.13, and 14.15.

The main part of an LSD file contains declarations of classes, enumerations, and types.

14.5 Class Description

Ordering of classes may be nearly arbitrary, since a forward declaration is generated for each <class>. The only rule is that the base class must precede it's children.

Element <class> has the following attributes:

- name (required): An identifier for the class.
- base: Name of a base class. Can be omitted if a default base is declared in header part of the document. Multiple and virtual inheritance is not supported.
- abstract: Contains "yes" or "no", default is "no". Abstract classes can't be instantiated and lack factory methods. A class containing an abstract method must be explicitly marked as abstract.

Element <class> may have the following content:

- <field>: Describes a data member. Visibility of each data member is private. set(), get(), and alt() methods are public. <field> attributes are:
  - name (required): Field identifier.
  - type (required): Data type. If the member is a class, there should be simply the identifier of the class. The generator will automatically create a smart pointer and operate with it. See also Garbage collection 14.10 below.
  - set: May contain "special", "none", or "". Default is "". By default, the set() method is generated and trivially returns the field value. "none" means that the member lacks set() method. And "special" means that the set() method is present but hand-written. See also Class data access 14.6.
  - get: Describes the get() method in analogy to set().
  - alt: Describes the alt() method in analogy to set().
  - specifier: Contains a specifier for the field, allowed values are "static" or "", default is "". A static member is treated specially as C++ requires. (It's declaration is placed to the bottom of .g.cc file.) The init attribute works well with this.
  - init: Contains an initializer for the field. May contain simply a value or "" or "void", default is "void". "void" means that the member is always initialized through a parameter of a factory method. Otherwise, the attribute value is used as an initializer. See also Creating an instance 14.12.
  - check: Value of each field is assertion-checked with a special macro before class construction (see 14.12). The default name of the macro is "checked" or may be specified in <default-check/> element (in LSD header part 14.4). Setting the check attribute to a field allows you to call a different macro.
- <collection>: Describes a collection of values, similarly to the <field> element, using the same attributes. One difference is in the init attribute, which is restricted to two cases. 1) Missing init or "void" means a collection initialized by a factory method parameter. 2) init="" means initialization as an empty collection. Attribute kind may be "list", "vector", "set", or "map". It's default value is specified in <default-collection> top-level element. Lestes STL wrappers 16 are always used to ensure garbage collection. The map collection has two additional attributes:
  - key (required): Contains a data type used as a key in the map.
  - comparator: Describes a special comparator for the keys. For example:
    comparator="::std::greater<int>"
- <method>: Describes a method that will be written by hand. The attributes are:

- name (required): Method identifier.
- type (required): Return type. (See Garbage collection 14.10.)
- qualifier: A qualifier for the method. For example "virtual".
- specifier: A specifier for the method. For example "static" or "abstract".

An abstract method must have qualifier="virtual" and specifier="abstract". Such a class must have abstract="yes". Factory methods 14.12.2 are not generated for abstract classes.

Parameters of the method may be given as a sequence of <param> elements into the <method> element. Each <param> has two required attributes - name and type.

- <typedef>: Describes a type definition used in this class. It's visibility is public. See also Type description 14.8 below.
- <enum>: Describes an enumeration type used in this class. It's visibility is public. See also Enumeration description 14.7 below.
- <visitor>: See Visitors 14.14.

14.6 Class Data Access

All data members of a generated class are **protected**. By default two special methods are generated to realize access to such encapsulated fields. Visibility of these methods is **public**. Each of them may be user-defined or missing. For the field x of a type T the methods are:

- T x_get() - Returns the value of the field.
- void x_set(T a) - Assigns a given value to the field.

14.7 Enumeration Description

On the top level or in a class definition, there may occur an <enum> element which defines a new enumeration type. It has one required attribute **name** with an identifier of the new type. <enum> contains a sequence of <e> elements representing enumerators, each of them having a required attribute n (as 'name') with enumerator identifier. Optional <e> attribute val associates the enumerator with a constant expression. See also an example in the C++ Standard **[7.2/2]**.

14.8 Type Description

On the top level or in a class definition, there may occur a <typedef> element which defines a new type. It has just two required attributes: name with the new identifier and type with a type specification.

14.9 Doxygen

Doxygen http://www.stack.nl/~dimitri/doxygen/ is a program used to generate documentation directly from the source files. Hand-written parts of such a documentation are placed into the source files as special comments describing the nearby objects. Lestes Structures Generator allows users to insert such comments into the generated C++ files.

Almost anywhere in the LSD file, there may occur a <dox> element to add a Doxygen-like comment. It contains two elements: <bri> with a brief description and <det> with a detailed description. The <det> element can be omitted, <bri> is required. The <dox> element has no attributes (except the top-level <dox>, see LSD header part in 14.4).

```
<dox>
        <bri>Brief comment</bri>
        <det>Detailed description</det>
</dox>
```

The comment describes the element it occurs in. These LSD elements may contain <dox>:

```
<typedef/>
<enum/>
<class/>
<field/>
<collection/>
<method/>
```

The `method/dox` element should not contain `<det>` since detailed description of a method should be placed near it's implementation (in hand-written `.cc` file).

Also the abstract visitor classes written by the LSG may be commented. In LVD files the `<dox>` element may occur in these elements:

```
/lvd
/lvd/visitors/v
/lvd/visitors/v/special
```

It is used the same way as in LSD files.

14.10 Garbage Collection Support

The C++ language lacks automatic garbage collection and forces programmers to treat each instance of a class to be deallocated. It is however very difficult in a project like Lestes with about one thousand of different classes and huge heterogeneous data structures. It was found that a garbage collector is necessary. It's interface and usage is described in Garbage collector 19.1.1 chapter. Users have to use two types of smart pointers everywhere and to write a marking routine for each class. The generator strongly helps to realize these two duties.

14.10.1 Marking Routine

The `gc_mark()` method is generated fully automatically for all LSD classes (including abstract ones). The visitor classes generated from LVD files are also made via LSD (see Visitors 14.14), so they have their marking routine generated the same way.

It's visibility is always **protected** and the prototype is common for all classes:

```
virtual void gc_mark();
```

The routine simply calls `gc_mark()` routines of all garbage-collectible members specified in this class and of the base class. There is no need to touch inherited members. And there is no need to iterate through collections of garbage-collectible types, since the STL wrappers do it themselves.

14.10.2 Smart Pointers

The garbage collector forces users to use two kinds of pointers represented by `ptr<T>` and `srp<T>` templates (both point at a type `T`). The `srp` pointer must be used for members of a structure, class, or collection. The `ptr` pointer everywhere else (return values, local variables, method arguments, etc.). The generated code obeys these rules.

14.10.3 Garbage-collectible Types

Garbage-collectible types must be handled by smart pointers, non-collectible types are handled by value. The rule is that all classes are derived from `::lestes::std::object` and are garbage-collectible and all simple types are not collectible. (A few exceptions exist but has no effect on generator.) The generator tries to automatically decide whether an identifier belongs to a class or simple type but sometimes it needs a hint.

The basic assumption is that unknown types are not collected. Types like `lint`, `ucn_string`, `character`, etc. are not described in any LSD file and the generator doesn't make pointers on them. The user must let the generator know about each violation of this assumption. Note that `<include>` tags are not used when searching for "known" classes, their purpose is different. For classes used in an LSD file (as a base, as a member, as a method argument, as a visitor type, or so) but declared somewhere else, there must be one of these marks:

```
<forward-class name="thomas">
<using-class name="alva" packages="::lestes::std::"/>
<foreign-class name="edison">
        <p>lestes</p>
        <p>std</p>
</foreign-class>
```

The naming and syntax of these marks has historical reasons and shouldn't be taken too seriously. May be a subject to change in the future.

The `using-class` element declares a class that is used as a base class, as a class member, as collection member, as a method parameter, return type, or so. If it lies in a different namespace, the attribute `packages` is filled, otherwise it may be omitted. No special C++ text about such a class is generated, except the namespace qualification. If nothing special is needed, use `using-class`.

The `forward-class` element declares a class as `using-class` does. The difference is that `forward-class` causes a C++ forward declaration to be generated at the beginning of the .g.hh file. For example:

```
class thomas;
```

The class must be in the same namespace as the LSD file contents.

The `foreign-class` element does the same as `forward-class` but the class lives in a different namespace. The sequence of `<p>` elements denotes the namespace. A forward declaration is generated with respect to the namespace policy and it's placed out of the binding `package/end_package` braces of the generated file. Example:

```
package(lestes);
package(std);
class edison;
end_package(std);
end_package(lestes);
```

There are some tips about using these tags in the LSD FAQ 14.15.

14.11 Dumping

LSG inserts some code and data into each class for purposes of the dumper. Static member `reflection` contains metadata about the class - it's name and it's members names. The metadata are created on demand and only once by the `reflection_get` method. The dumper also needs concrete values of the members. Method `field_values_get()` returns such list of values.

14.12 Creating a Class Instance

14.12.1 Constructor

Each generated class has just one constructor and it's visibility is **protected**. It has one argument per each member of the class including inherited ones. The values passed to the constructor are used to initialize all the members. If some of the data is passed by pointer, the pointer is checked for non-null-ness. Passing a `NULL` pointer causes an abnormal program termination. The goal is to avoid creation of incomplete classes that could cause bugs hard to find. But sometimes it can be very annoying.

14.12.2 Factory Method

Class instances in Lestes are almost every time made by so-called factory methods. It is a static method called `create()` that allocates and initializes a new instance. For the purposes of initialization (see above) the factory method needs a value for each member of the class. One way to do it is to have one parameter of the factory method for each member (as it is by the constructor). This case is always enabled, since such a `create()` method is in each generated class that is not abstract. But we often don't want to fill all the members by hand when creating an object. Some fields may have initial values specified in the `init` attribute. In such a case the `create` method is overloaded and may have a different number of parameters. The second one lacks parameters for those fields that have the `init` attribute specified. If each member has an initial value, there will be a `create()` method without parameters.

14.13 Namespaces

The `package` and `end_package` macros behaves like left and right braces. The main content of an LSD file is placed into one namespace in both the `.g.hh` and `.g.cc` file. Including the contents of `hh-header`, `hh-footer`, `cc-header`, and `cc-footer` elements. A class from a different namespace must be declared with `using-class` or `foreign-class` and then the namespace prefix is automatically placed before each occurrence of it's identifier. For example the `object` base class should always have a declaration like this.

```
<using-class name="object" packages="::lestes::std::"/>
```

Only the included files and forward declarations of foreign classes are placed outside the main package braces.

14.14 Visitors

14.14.1 Basics

Visitor is a design pattern, see Gamma et. al, Design patterns. It's a master peace of object-oriented design in the Lestes project. The goal is to separate actions above the complex structure from classes that contain the data. Visitor is a class representing an action that takes place on classes from a specific set. The set is determined by a name of one class and it includes the class together with all non-abstract classes derived from it. Abstract classes cannot be instantiated and therefore cannot be visited. The root class has a special note in it's LSD description. For example:

```
<class name="as_expression" abstract="yes" base="as_base">
        <!-- some content -->
        <visitor name="as_expr_visitor" type="void"/>
</class>
```

This defines a visitor class **as_expr_visitor** that can perform an action on any non-abstract class derived from **as_expression**. The class itself is abstract so it cannot be visited. The action represented by a visitor has a result of some type. In the above example the type is **void**.

A visitor is used every time when an action implementation depends on the class that it's performed on (the **visitee**).

14.14.2 Abstract and Concrete Visitors

The LSG is able to make a visitor class specific for a sub-tree of the class hierarchy. Such a class is always abstract, because the generator has no information about the action implementation. And note that there can be more than one action performed on the same set of classes. It would be nice to have a reusable abstract class to visit a set of classes and the concrete visitors (actions) would be derived from it. But different actions have different types of results, so we don't know what visitor type to specify in the abstract class. This is the reason why we have later decided to prefer visitors with return type `void`. If there is a need to return a value, it can be stored into a special member in the concrete visitor class.

14.14.3 What Does the Generator Produce

Each class derived from a class with visitor has a generated method with such a prototype:

```
virtual T accept_V( ptr< V >);
```

where `V` is the visitor's name and `T` is it's return type. If the class is declared as abstract, the `accept()` method is also abstract. Otherwise an implementation is generated into the `.g.cc` file. It simply calls corresponding `visit()` method of the visitor class.

The abstract visitor class is named as specified and it is derived from `::lestes::std::visitor_base`. It contains a public abstract method for each visitee.

```
virtual T visit_A( ptr< A >) abstract;
```

where **T** is the action result type and **A** is the visitee. The **visit** method represents an action implementation on a specific class.

Some other stuff is generated by LSG to make the user's life easier. See Implementing visitors 14.14.6.

14.14.4 What Does the LSG User Have to Do

Simply said, the user has to derive his concrete visitor from the generated abstract class and implement all the `visit*` abstract methods. If some data storage is needed for the implementation, it can be added to the concrete visitor class. (For example to store a result or whatever.) To perform the action on a given object, instantiate the concrete visitor and call the visitee's `accept` method with pointer to your visitor.

Note that the abstract classes are generated into a *.v.lsd file in the LSD format (by `lvd2lsd.xslt` stylesheet). This allows users to derive their concrete classes via LSD with all the comfort (automatic dump support, marking routines, safe constructors, factory methods etc.).

14.14.5 LVD Files

The abstract visitor classes are generated by a mechanism completely different from LSD processing. A special XML description of what to do is needed. The header part of an LVD file is very similar to LSD except the different XML namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<lvd xmlns="http://lestes.jikos.cz/schemas/lvd">
        <dox><bri>Some abstract visitor classes</bri></dox>
        <file-name>zzz</file-name>
        <imports> . . . </imports>
        <implementation-imports> . . . </implementation-imports>
        <packages>
                <p>foo</p>
                <p>bar</p>
        </packages>
        <hh-header/>
        <hh-footer/>
```

A sequence of <uses> elements follow such a header. The element has a **href** attribute with a path to an LSD file, where some visitees are defined. The main part is in the <visitors> element that simply specifies names of the visitors.

```
<visitors>
        <v name="quo"/>
        <v name="vadis"/>
        <v name="domine"/>
</visitors>
```

All other necessary information is automatically searched in the LSD files via `<uses>`. Usage of Doxygen comments is described in the above section 14.9. Instructions for `<import>` are in LSD FAQ 14.15.

Note: Do not accumulate all visitors into one LVD file to avoid include hell. 4 of 5 Lestes developers recommend: One algorithm, one file with visitors.

14.14.6 Implementing Visitors—The Inheritance Tree Cuts

Some visitors may visit a lot of classes. In practise, the action implementation is often the same for many of them and it would be annoying to write all the `visit` methods by hand. Any small change in implementation would force user to type it repeatedly - a possible source of bugs. So which classes do have the same implementation?

Let's watch a real-life example `http://lestes.jikos.cz/html/classlestes_1_1lang_1_1cplus_1_1sem_1_1ss__type.html`. Class `ss_type_visitor` is an abstract visitor for `ss_type` descendants. The action depends on a specific class. It is different for `ss_array`, `ss_enum`, `ss_function`, `ss_builtin_type`, `ss_struct_base`, and so on. But it doesn't depend on a concrete builtin type (15 different non-abstract classes!). And it's also the same for all descendants of `ss_struct_base`. We would therefore like to implement it just once for all `ss_builtin_type` descendants. And the second time for `ss_struct_base` descendants. The LSG offers such a feature.

```
<v name="ss_type_visitor" >
        <special name="ss_type2info_base">
                <cut at="ss_struct_base" method="process_ss_struct_base"/>
```

```
                <cut at="ss_builtin_type" method="process_ss_builtin_type"/>
        </special>
</v>
```

In this example the LSG will generate an abstract class `ss_type_visitor` as usually. In addition, there will be a class `ss_type2info_base` derived from `ss_type_visitor`. All its `visit` methods for `ss_builtin_type` descendants are automatically implemented as a call to a `process_ss_builtin_base` method. For descendants of `ss_struct_base` the method `process_ss_struct_base` is called. Both `process*` methods are automatically added as abstract, since the generator doesn't know about the true action. Therefore the class `ss_type2info_base` is also abstract. The concrete visitor is derived from it and implements the two `process*` methods and all the `visit*` methods that remain. Such a concrete visitor is `ss_type2info`. Its hand-written implementation is in `ss_type2info.cc` file and it contains just one method for each nearest descendant of `ss_type`, all in all 13 methods. That was our goal. Note that `ss_type2info` visits 28 non-abstract classes. And note that there is no problem to have more than one `<special>` in a specific abstract visitor.

One additional feature of the generator is used in the above example and not yet mentioned. If a class implements some inherited abstract methods, C++ requires explicit declaration of all such methods in the class declaration (header file). It's easy to accidentally omit one of the declarations. Then the class remains abstract and the compilation fails at it's instantiation. In the worse case some declaration may be extra and there remains an extra implementation that is never called. Both these problems may be prevented using automatic generation of the declarations. Note that the class `ss_type2info` has a special line in it's LSD definition.

```
<declare-abstract-methods prefix="visit_" skip-defined="yes" />
```

It generates declarations of all abstract methods in the class that start with **visit_** prefix. Instead of the prefix a suffix may be specified with `suffix="fff"` or an exact match may be required with `exact="identifier"`. The `skip-defined` attribute may be optionally used to skip the methods that have been defined previously. Omitting all of these attributes causes all abstract methods to be declared and forced to be implemented.

14.15 LSD Frequently Asked Questions

14.15.1 Using-class or a Forward Declaration?

In most cases the forward declaration is not necessary and the `<using-class>` element is enough to let the generator know about the class. When do we really need a forward declaration?

- For each visitor class.
- When `alice.lsd` uses a class `photon` defined in `bob.lsd`, but `bob.lsd` also uses some classes from `alice.lsd` and therefore needs to import `alice.g.hh`.
- ?

14.15.2 Import or Implementation-import?

In many cases the `<import>` isn't necessary and only `<implementation-import>` should be used to avoid include hell. But sometimes you need to see the whole declaration of a class in the .g.hh file.

- The basic rule is that you need to `<import>` all classes used as bases. `<implementation-import>` should be enough for everything else.
- When gcc gives you a strange "invalid static cast" error in the ptr.hh file, it is most likely that the class declaration isn't visible in the `.g.hh.` file. `<implementation-import>` of an appropriate header should fix it.

Chapter 15

Build System

Build system

15.1 Basics

There is only one 'main' Makefile, in which everything is implemented. This Makefile resides in build/Makefile, and all other Makefiles in the tree should include it using '../..' etc. as relative path. This is done to allow invocation of make on any arbitrary level, not only on top-level (which could fail in some cases, as cross-tree dependencies on generated files don't work - see below) and the top-level Makefile must be included from all Makefiles

When invoking make, the default mode of operation is 'quiet', it means that make gives only basic and short information on what it is currently doing, but doesn't print messages about Entering/Leaving directory.[1] The verbose mode of build process can be turned on by specifying parameter V=1 to make. Example:

```
$ make generate V=1
```

- Makefile prints paths relative to the project root directory when informing about currently processed file. To make it print only the filename, without the path, specify non-empty S parameter. Example:

```
$ make S=1
```

15.2 Supported Targets

- `<none>` (aliased by 'all' target) - if make is invoked without any specified target, the following phases/targets are invoked (in this particular order) (for the description of each phase/target, see below)
- `predepend-recursive`
- `generate-recursive`
- `depend-recursive`
- `compile-recursive`
- `link-recursive`

Description of individual phases follows.

- `predepend-recursive` - this is to generate first Makefile.dep, which is needed even before generated files are created. This generates dependencies of lvd and lsd files
- `generate-recursive` (aliased by 'generate' and 'gen' targets) - this target creates generated (not compiled) files from sources specified in SFILES variable (see below). It generates
  - .tab.cc from .y

[1] If one whishes to see these and just these messages, but the rest of the build process should stay in quiet mode, -w parameter to make (which is internal make parameter, not implemented by our makefiles) should be used.

  - .yy.cc from .l
  - .g.cc from .lsd
  - .g.hh from .lsd
  - .m.cc from .lmd
  - .m.hh from .lmd
  - .v.cc from .lvd
  - .v.lsd from .lvd

- `depend-recursive` - this target is run after the generate phase, to be able construct dependences of sources on generated sources, which must exist in the time of dependencies generation. It uses both GNU makedepend and our makedepend.xslt
- `compile-recursive` (aliased by '`compile`') - .o objects are generated from sources
- `link-recursive` - linkage of tests (see below) and objects (see below) and default objects (see below) is performed to create executable binary.
- `clean` - files from current directories (and recursively from directories in `DIRS` variable (see below)) are deleted. The deletion is performed on files which were generated and compiled by generate and compile phases (see above). This is computed from `SFILES` and `SFILES_TEST` variables from Makefile in current directory. Files specified in `CLEANFILES` are removed as well. All the other files are left untouched.
- `mrproper` - same as clean, but more aggressive - in addition to what clean does, mrproper erases all "*.o *.tab.cc *.yy.cc *.v.cc *.v.lsd *.g.hh *.g.cc *.mdg.cc *.mdg.hh core* Makefile.dep" in current directory.
- `test` - compiles and links files specified in `SFILES_TEST` variable (see below) to make executable binaries, and then runs them. This is used to test functionality of the thing you have implemented. During this phase, also all files on which this test creates (particularly files implementing the feature it is testing :)) are automatically generated/compiled (based on generated dependencies).
- `doc` - this is quite special target, as it is intended to be called just from top-level directory, and is used to generate documentation in `doc/doxy/html` directory from the `doc/doxy/Doxyfile` configuration file. In directories other than top-level directory this shouldn't be invoked, as it would probably fail, not finding directories it is expecting.
- `doc-clean` - wipes out the whole doc/doxy/html directory

15.3 Variables in Makefiles

Following is the list of variables on can use in per-directory Makefile and explanation of their meaning.

- `DIRS` - list of directories for recursive descent of make process
- `SFILES` - should contain all filenames, which serve as 'sources', either for compilation or generation. The proper action (compile/generate) is derived automagically from the file extension.
- `SFILES_TEST` - list of files that should be compiled only if 'make test' is issued. Sources specified in this variable are linked into executable binary (for linkage process, see below).
- `CLEANFILES` - list of additional files that should be removed when performing 'make clean' or 'make mrproper'. Typically used for files that are output by tests, as they have arbitrary names and thus are not covered by any of the usual rules.
- `TESTS` - specifies which of the binaries, coming from compilation of files in `SFILES_TEST`, should be run and their return value checked. If you wonder in what situation file could be in `SFILES_TEST` and not in `TESTS`, then the answer is that it could be if you want to test the linkage of things together, but you are not interested in running the resulting binary itself.
- `TESTS_FAIL` - basically the same meaning as `TESTS`, the only difference is that the "correct" run of tests listed in this variable is considered a situation when the executable returns non-zero return code.

- `TESTS_FAIL_SILENT` - basically the same meaning as `TESTS_FAIL`. The difference is, that output of running these tests is redirected to `/dev/null`. The motivation for creating such a test was that some tests, when failing, are producing non-trivial amounts of output data (stack traces, error messages, etc), but the person running `make test` is typically interested in succeeded/not succeeded information, i.e. the return value.
- `RUN` - specifies executables which are to be run in the test phase of build process. These files are neither generated, compiled, nor cleaned in cleaning targets. It's main purpose is to be allow for running of helper shell scripts, which are executing compiled tests in some more sophisticated way.
- `MD_CC_FILES` - list of files (*.mdg.cc) which are generated in quite different way from all other generated files (lsd/lvd). These files are generated by running xsltproc on machine_description.md (in proper directory) and *.mdg.cc.xslt stylesheet (in proper directory). The main biggest difference from lsd/lvd is, that the lsd/lvd file is not in current directory, but files are generated into current directory.
- `MD_HH_FILES` - same as previous variable, just for .mdg.hh files

15.4 Variables Used for Linking

- `EFILES` - the list of executable files that should be created as the result of linking compiled object files. Which object files is specified by adding '.objects' suffix to the name specified in this variable (see example below). The path must be absolute from the top-level directory of the project. Outside of the objects specified in <executable>.objects variable, there are some default objects, linked to every executable by default. The list of these objects can be obtained by looking at variable `DEFAULT_OBJECTS` in build/Makefile
- `EFILES_TEST` - the same as previous one, but for the `make test` case.

15.5 Example Makefile

This makefile

- descends recursively into subdirectories named 'subdir1' and 'subdir2'
- generates .hh headers from source3.lsd and source4.lvd
- generates .cc sources from source2.l source3.lsd source4.lvd
- compiles source1.cc all .cc files coming from 'generate' phase into objects of the corresponding names
- links lestes/lang/my/current/directory/source1.o and lestes/lang/blah/foo/bar/some/object.o and objects specified in DEFAULT_OBJECTS in main Makefile, into executable 'source1'.

When issued as 'make test', after resolving dependencies, it does the following:

- compiles source1.test.cc into source1.test.o, when issued as 'make test'
- links source1.test.o with source1.o and objects specified in DEFAULT_OBJECTS in main Makefile
- runs ./source1.test

```
DIRS = subdir1 subdir2
SFILES = source1.cc \
source2.l \
source3.lsd \
source4.lvd
SFILES_TEST = source1.test.cc
TESTS = source1.test
EFILES = source1
source1.objects =
lestes/lang/blah/foo/bar/some/object.o \
lestes/lang/my/current/directory/source1.o \
source1.test.objects = lestes/lang/my/current/directory/source1.o
BUILDMAKEFILE = ../../../../../build/Makefile
include $(BUILDMAKEFILE)
```

Chapter 16

STL Wrappers, Smart Pointers

To make STL collections cooperate with the garbage collector in lestes, we had to wrap the STL classes. Technically, we "just" inherit from both our garbage-collectible root class (object) and the standard STL collection class.

This should preserve the standard API of STL containers. Few exceptions exist, though.

16.1 Supported Classes

- `list, vector, map, set, stack`
- `pair` - see section about `pair` below

Each is declared in its own header file, lestes/std/{class_name}.hh .

16.2 API

The lestes collection templates have the same names as their STL counterparts, just the namespace is different: `::lestes::std`. Care must be taken when instantiating the actual collection class.

When declaring collection of garbage-collectible items (of type T), the actual type passed to the collection template must be `srp<T>`, not `ptr<T>`, as the pointers that will reside in the collection behave just like structure pointers - they are by no means root pointers (`ptr`).

In lestes, one does not use constructors, but factory methods (`create`) to get an instance of a class. Collections are no exception. To keep our wrappers simple enough, only two create methods are supported:

- `create()` - creates an empty collection
- `create(ptr)` - takes a pointer to the same type of collection, creates a shallow copy

Construction from iterators is not supported, as it turned out to be a problem for all the compilers we have tested.

The rest of the standard API is not touched, so everything from iterators to algorithms works as expected. See notes below for pitfalls.

16.3 Notes

To correctly interact with garbage-collector, the collection templates have to "know" whether given argument denotes a simple or garbage-collectible type. It is rather easy to find out - any non-simple type T must be passed as `srp<T>`. That also means that specializations for every possible combination of simple and non-simple arguments of the template must be written, however.

Because of the dumper, even simple types are special in a way. For every simple type `S` that can used as an argument to a collection template, specialization of the `objectize` template must be defined. Specializations for the most used simple types, eg. `lint` and `lstring`, are done and are supported in the dumper code.

Specialization for enumeration types can be defined by a macro. It treats the value as an signed integer (`lint`).

Finally, there is the need for collections of items of types that are not garbage-collectible, but are not worth dumping. Another macro provides objectize specialization that makes the type supported by the collection templates, yet not dumpable. See the dumper documentation for details.

16.4 Pair

The `pair` template is not a collection, but must follow similar rules with regard to dumper and garbage-collector.

Various standard STL functions return pairs as their result, eg. `map::insert()` does. For the garbage-collection to work, standard `::std::pair` must not be stored on the stack or in a class. Use `::lestes::std::pair`.

Unlike collections, `::lestes::std::pair` has not only the empty and the copy-constructor-like create method, but it can also create itself from `::std::pair`. Correct usage should read

```
using namespace ::lestes::std;
typedef map<object,ulint> map_type;
typedef pair<map_type::iterator,bool> xpair_type;
// in method body:
ptr<map_type> a_map = ...
...
ptr<xpair_type> result = xpair_type::create( a_map.insert(...) );
```

Note that `map_type::iterator` must be marked as nondumpable, see "notes" section of this file or lestes/std/objectize_macros.hh .

Chapter 17

Dumper

17.1 Goals

The goal of the dumper code was to be able to easily dump any live structure to XML (or any other format). We did not want "perfect" reflection support, nor the most generic one, but one that can will cover the usage patterns of our project. Also we wanted to keep the code as simple as possible.

Vast majority of classes are generated from XML, so once the support for the dumping metadata is integrated within the generator, almost everything is instantly dumpable.

There is a standard dumper in the lestes project.

- `::lestes::std::dumper::dump( const ostream s, ptr<object> o )`

The function dumps every object reachable from pointer o to given standard `::std::ostream s`. It retuns s. The output is valid XML and is not meant to be read by humans. It is "flat", its depth is constant and very small.

- `::lestes::std::readable_dumper::dump( const ostream s, ptr<object> o )`

Has the same semantics as the official dumper, but the output is much more readable. The format is XML, it would be very difficult to validate it however. It is supposed to be used until a good visualiser for the official dumper output is written. It primarily uses element nesting and the depth of its output is not limited. It is not very robust, as it might run out of system stack.

Both dumpers are declared in `<lestes/std/dumper.hh>` .

17.2 Achievement

The primary goal was fulfilled. The dumper works well.

The final code is simple enough, too. We avoided member pointers, which costs us a little on the performance side, but that was not our priority.

17.3 Assumptions

From the point of view of the dumper, every dumpable instance has

- one or more sub-instances of base classes, each having
  - a name
  - zero or more own fields, each having
    - name
    - type
    - zero or more values
- exactly one base class

`lestes::std::class_reflection`

- stores this meta-information about a single sub-instance ("I")
  - name of its actual type ("I")
  - names and types of fields that are declared inside C, ie. not the inherited ones
- can be computed (created) only once (and stored in a static class member) for every class, including collections - they use multi-value fields

Field value is always a pointer to an instance - for the dumping purposes, simple type fields are wrapped in the `objectize` template; specialization for every possible simple type must be written by hand (`ulint`, `lint`, `ucn_string`, `ucn`, `bool` are provided; `enums` are cast, see below)

For the dumper, collections behave just like non-collections

- standard collections have one field: "item"
- there are multiple values for that field, however
- dumping support for every collection must be written by hand

17.4 API

`class_reflection` (implemented in lestes/std/reflect.hh)

- stores only the meta-information about a single (class)
  - name of the sub-instance
  - (name,type) pair for each field declared in the very class
- does not store any link to the base class
- `lstring name_get()` returns name of the actual type
- `field_metadata_get()` returns list of `<lstring,lstring>` pairs containing a (name,type) pair for each of the non-inherited fields
- has a standard `create()` method

Dumpable class C must

- be derived from `lestes::std::object`
- override virtual method `reflection_get()`
  - this method returns list of `class_reflection`, each describing a class in the hierarchy from object to the actual class of the instance
- override virtual method `field_values_get()`
  - this method returns a list that - for each of the fields, `inherited included` - holds a list of the values for that field
  - the order of the value list is the same as the corresponding pairs in `class_reflections` returned by `reflection_get()` for the very instance; values from base classes precede values from derived ones

`dumper_visitor` (implemented in lestes/std/dumper_visitor.hh) is an abstract visitor class. There is the need to distinguish between simple and non-simple types when dumping the actual values.

`::lestes::std::object` has virtual method `accept_dumper_visitor()` that takes a pointer to `dumper_visitor` and calls its `visit_object()` virtual method, passing `ptr<object>(this)` as its only argument. Simple type wrappers override the `accept_dumper_visitor()` method and call different `visit_AAA()` methods - this way a dumper may distinguish between real pointer values and "fakes" - wrapped simple types

Currently, visit_AAA methods for these types are declared:

- object

- lint
- ulint
- bool
- lstring
- ucn_string

There is one special visit method: `visit_nodump()`, used for types that are not meant to be dumped.

`objectize` (implemented in lestes/std/objectize.hh) is `template<T>` that wraps simple types in a `ptr`. The default one does nothing, one must write a specialization for every type that needs to be supported. It has a `create` method that takes [reference to a] value of type T. enums are special, as we do not want to have a visit method for every possible enum that is used throughout the project

- for simplicity, enums are output as lints
- however, `objectize` must be specialized for the enum type nonetheless, to preserve the type safety that enums bring
  - use specialize_objectize_for_enum() macro found in lestes/std/objectize_macros.hh; the create method of the resulting specialization of objectize just creates objectize<lint> with the enum value cast to a lint
  - all the enum types are then handled as lint, for simplicity

Support (specialization) for non-garbage-collectible types, that are never meant to be dumped is provided

- `specialize_objectize_nodump(), specialize_objectize_nodump_reference()` macros in lestes/std/objectize_macros.hh
  - for reference types, use the latter one; the former one is sufficient for all the other cases
- the produced specialization calls `visit_nodump()` in its `accept_dumper_visitor method`
- this makes collections of iterators possible, for example

The API was carefully designed to permit the implementation of a dumper yet to prevent other uses.

17.5 How a Dumper Examines Given Instance

The dumper always examines a pointer (this makes `NULL`s easy to detect and handle in a special way). To find out whether given pointer points to a "real" class, or is just a wrapped value of a simple type, `dumper_visitor` is used. Examining non-null pointers to "real" instances is the core of the dumper.

- `reflection_get()` retuns a class_reflection for every base class, starting at object, ending at the actual type of the instance
- `field_values_get()` returns a list which for each field hold a list of the values of the field; inherited fields are also included
- now, for each of the metadata pair deep in the `class_reflection` list, there is exactly one list of the values belonging to the field; the rest is just iterator work .

17.6 Examples

See lestes/std/dumper.test.cc for examples of writing dump support for a class by hand; the file also contains a DFS dumper. lestes/std/dumper.?? contain official dumper, that uses BFS and produces "flat" XML output.

Chapter 18

Logger

The logging facility is meant to be used for debugging output only. It was not designed to be used to report errors or warnings to the user.

The main goal was to allow turning parts of the logging messages on and off without the need to recompile the whole source tree. This is achieved by storing the configuration for the logging objects in external XML file. This solution was inspired by the java log4j library.

18.1 Usage

All the logger classes are located in the `::lestes::msg` namespace. The following text does not qualify these classes. They are declared in header file <lestes/msg/logger.hh>.

The whole logging facility consists of a tree of instances of `logger` class. Each of the instances can be individually turned on or off. There are no priorities, as it would not be clear which priority to use for a particular message. Furthermore, the priorities can be easily "emulated" by a trivial subtree - one with no branches.

The root `logger` instance is created automatically by the logger code and can be accessed by calling `logger::root_instance()` method. It has empty name and is parent of itself.

When creating a logger instance, a name and a parent must be specified. No two children of a logger can have the same name and every logger must have a (non-NULL) parent.

Once the logger is configured (see below), to get the right ostream, one has to call operator << with a logger_formatter as its argument. The formatter's format method is called with pointer to the logger in question, and the ostream as its arguments. The formatter usually prints the name of the logger to the stream and returns it.

There are a few formatters prepared. Probably the most used one will be `fullname_formatter`. It prints full name of the logger (e.g. '`/syn/manager/commit`') followed by a colon and returns the passed ostream. It is used by the `llog` macro. Other formatters are also wrapped in macros as macros simplify formatter usage.

To simplify using the loggers and formatters, several macros exist. All of them take at least one argument - pointer to a logger. They always return an ostream, so that operator << can be used directly on their result.

- `llog( a_logger )` prints full name of given logger
- `llog_plain( a_logger )` does not print anything, returns the ostream untouched
- `llog_xml_open( a_logger, tag_name )` opens an xml tag with given name, it will have attribute named '`by`' containing full name of the passed logger
- `llog_xml_close( a_logger, tag_name )` closes given tag; just a wrapped around `llog_plain`

For an example, see `lestes/msg/logger.test.cc` file.

18.1.1 An Alternative Approach to Logger Usage

Each logger can have one formatter, which can be used as a default formatter for the logger. There formatter is obtained and changed by getter and setter of the `formatter` field.

To use the default formatter a pointer to the affected logger is used as the left argument of the << operator. Assuming that the right operand is *x*, the expression is reduced to `*logger << logger->formatter_get() <<` *x*. In order to invoke the formatter's end action each use should be terminated by using `<< ::lestes::msg::eolog` as the last element of the logger invocation.

For an example of the alternative approach, see `lestes/msg/logger_util.test.cc` file.

18.2 Configuration

The loggers are turned on/off by calling `logger::init()` method. It takes one argument, filename of the configuration XML file. The whole logger tree must be created before calling this method. It must not be called twice during one run of a program.

There is only one element allowed in the configuration file: `logger`. It has these optional attributes:

- self
  - allowed values: "on", "off"
  - if exists, turns this logger on/off
  - if does not exist, inherited value (i.e. value of 'children' attribute found in the closest parent element) is used to turn this logger on/off
- children
  - allowed values: "on", "off"
  - if exists, turns child loggers on/off, unless overridden by children and/or self attributes in child elements
- file
  - if non-empty, output of this logger and its children is sent to file with specified name
  - if empty or non-existing, standard error output is used
  - can be overridden in child elements
- formatter
  - contains the name of the default formatter. possible values are empty, `plain`, and `simple` to use the plain formatter, `fullname` to use the fullname formatter, `color` or `colorific` to use the color formatter, which outputs terminal escape sequences that color the output
  - alternatively a list of such formatters can be specified surrounded by parentheses and separated by spaces; the so named formatters are applied in order; eg. `(colorific fullname color)`
- parameter
  - this attribute contains a string that is passed to the formatter during its creation; it is generally not used except for the colorific formatter, where it specifies the requested color from `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`, optionally prefixed by `bright` or `bright-`.
  - if the formatter is specified by a list the parameters to each constituent can be specified in a similar list; eg. `(red nothing blue)`

(1) The root element must contain both 'self' and 'children' attributes as it is not a child. (2) If more logger elements have the file attribute set to the same value, both corresponding logger instances write to the same file. Note that the comparison is based purely on the attribute value. This means that "a.log" and "./a.log" are considered to be different, for example.

To close the files opened by `logger::init()`, `logger::cleanup()` must be called.

A skeleton configuration file can be produced by calling `logger::dump_skeleton()` method. It creates valid configuration file with all the loggers turned off. The tree must be created before calling this method, obviously.

For an example, see lestes/msg/logger.test.xml file.

18.3 Implementation Notes

The configuration XML file is parsed by libxml2.

The children of a logger are stored in a map indexed by child logger name.

Files opened in `logger::init` are stored in a map indexed by the filename. Thus the limitation mentioned above.

Chapter 19

Memory management

19.1 Introduction

In a project using many interlinked dynamically allocated data structures, which change rapidly during the execution, it is nearly impossible to release the structures manually. It is not easy to tell whether a certain structure can already be released. The structure can still be referenced and its elimination would cause runtime error when trying to access it. It is necessary to distinguish live structures from the unreachable to be able to release them. This burden is eliminated by introducing automatic memory management to take care of the bookkeeping.

19.1.1 Garbage collector

The memory management is performed by a garbage collector. The garbage collector operates on special *collectible* classes. It devises the lifetime of the dynamically allocated collectible classes upon certain properties. For correct operation of the garbage collector, it is vital that these properties remain consistent throughout the compiler run. To force this, it is necessary to obey several coding style rules and patterns when writing the source code.

19.1.2 Smart pointers

Bare pointers[1] to collectible classes must not be declared. Instead, special templated smart pointer types ptr<T> and srp<T> are used. The template parameter T designates the type of the pointee, which has to be a collectible class type. The ptr<T> types are used for local variables, function and method parameters, return values as well as static class and function variables and any global (namespace scope) variables. The srp<T> types are used only for non-static data members of collectible classes. It is not allowed to create ptr<T> dynamically, as it would most likely confuse the memory manager. It is also discouraged to create for example arrays of srp<T> dynamically. Instead, STL containers shall be used.

19.1.2.1 Operations on pointers

Pointers imitate the basic behaviour of bare pointers. All operations can be performed on pointers to arbitrary types, provided that the operation is correct for the bare pointers. It is also possible to have one of the operands of ptr type and the other of srp type, if pointee types are compatible.

It is possible to assign the pointers with operator= if both the types are the same, or the target pointee type is base of the source pointee type. Pointers can be compared through operator== and operator!= with usual meaning. Test for == NULL is done through conversion operator bool, which allows writing the if (x) condition. The operator! provides the != NULL test. The bare pointer can be obtained by dereferencing via operator*, fields of the pointee object can be accessed through the arrow operator->. Both these operators include internal check on whether dereferencing a NULL pointer. It is not possible to use the construct dynamic_cast<T>(x), converting x to the dynamic type T, on smart pointers, because the actual pointer is wrapped. Instead, the method x.dncast<T>() is called, which also checks internally, that the dynamic cast succeeded. The operator< is used for comparing the pointer values themselves, not the objects pointed to.

[1] To distinguish C++ pointers from the pointers used in the memory manager, we refer to plain C++ pointer as to 'bare pointer' and to the memory manager entity substituting bare pointer simply as to "pointer".

19.1.3 Collectible classes

Each of the classes managed by the garbage collector must be directly or indirectly inherited from the common base `::lestes::std::object` class. This declares the object of such class to be collectible. Each class inherited from a collectible class is automatically also collectible and cannot be excluded from the memory management mechanism.

19.1.3.1 Construction

Constructors of collectible must not be called directly, which implies that `operator new` cannot be used to create new instances. Instead, the instances are constructed by calling class static `create()` methods. Each `create()` method shall return a dynamically allocated instance of the class as `ptr<class_type>`.

```
ptr<class_type> class_type::create(bool a_param)
{
   return new class_type(a_param);
}
```

All user constructors shall be declared protected and used only in constructors of inherited classes and the `create()` methods. There is no imposed fixed relation between constructors and `create()` methods. It is possible to call the only defined constructor in all of the defined `create()` methods. The classes shall also declare a private copy constructor `class_type(const class_type &)`, but without defining its body. The same applies to the assignment operator `class_type operator=(const class_type &)`. The reason of such restrictions is to prevent operations which would interfere with the memory management mechanism.

19.1.3.2 Destruction

The destruction of instances of collectible classes is done automatically. It is only assured that it is done after the end of their lifetime. However, it is not stated when exactly will the destruction be performed and the destructor serves more as a finalizer. Thus it is recommended to omit defining a custom destructor entirely. Destructor must be defined if finalization is necessary to cleanup memory resources, which are not controlled by the memory manager, for example third party dynamically allocated classes. Destructor must be declared `virtual`.

19.1.3.3 Fields

Instances of collectible classes can have both static and non-static fields. The fields having non-collectible (or *simple*) types are declared in usual manner as

```
field_type field;
```

It is intentionally not possible to declare a field of collectible class type, these fields have to be smart pointers to the collectible classes. Non-static fields must be declared as

```
srp<field_type> field;
```

If these fields are not initialized with values passed into the constructor, they will be default-initialized to `NULL` and can have a value assigned later. Static fields must be declared as

```
ptr<field_type> field;
```

and can be initialized arbitrarily.

19.1.3.4 Methods

There is no special restriction on methods of collectible classes other than on normal functions.

Marking routine

To keep track of lifetime of collectible class instances, it is necessary to know the structure of the collectible classes. This is achieved dynamically, by a call to a method examining the instance. This method, inherited from `object`, is `virtual void gc_mark(void)` and each collectible class shall override it, unless it defines no fields of collectible types. The method body shall contain calls[2]

[2] Note that the method is called via dot so that it calls a `gc_mark()` method on the srp and not the collectible object itself.

```
field.gc_mark();
```

for all non-static fields of collectible types in the class and further a call

```
ancestor::mark();
```

for the direct base class `ancestor` of the class.

19.1.4 STL wrappers

In is not possible to use the default STL containers, which are not aware of the memory manager. Instead, collectible *wrappers* of the same names from namespace `::lestes::std` are defined for the basic containers. They can be used for all combinations of ordinary types and collectible types. These include

```
vector<T>, vector< srp<T> >
list<T>, list< srp<T> >
set<T>, set< srp<T> >
map<T, T>, map< T, srp<T> >, map< srp<T>, T >, map< srp<T>, srp<T> >
```

container templates. Each of the containers is declared in its own header file, lestes/std/{vector,list,set,map}.hh.

19.1.4.1 Construction

As for other collectible classes, the constructors were changed for `create()` methods. To keep the wrappers simple enough, only two create methods are supported:

- `create()` - Creates an empty collection.
- `create(ptr<T>)` - Creates a shallow copy of the same type of collection.

19.1.4.2 Destruction

The destruction of the container objects is fully under control of the memory manager. Once the container is not reachable, it can be released. Their collectible elements, if not referenced elsewhere, will also be released and their finalizers, if any, will be run.

19.1.4.3 Comparators

The default comparators for sorted containers `set` and `map` use the `operator<` to compare pointer values, which is perfectly legal for unique objects, but might not be appropriate for certain applications. It is, however, possible to specify a user defined comparator through a parameter to the respective container template.

```
typedef set< srp<some_type>, some_type_comparator > some_type_set;
typedef map< srp<some_type>, int, some_type_comparator > some_type_map;
```

19.1.4.4 Operations

Apart from creation, all operations defined on STL containers, work the same way as for regular STL containers. This applies also to algorithms and iterators. It is only necessary that the collection itself is referenced when working for example with iterators, as the iterator classes are not collectible

19.1.4.5 Pair

The STL `pair<T,U>` template, used for example in `map` must also be aware of memory management. Various standard STL functions, for example `map::insert()`, return `pair`, which can contain contain pointers to collectible objects. To keep the memory manager invariants, use of standard `::std::pair` to manipulate such values is forbidden. Instead, collectible `::lestes::std::pair` was defined. It can contain any combination of `srp` and simple types.

```
pair<T,T>, pair< srp<T>,T >, pair< T, srp<T> >, pair< srp<T>, srp<T> >
```

Unlike collections, ::lestes::std::pair has three create() methods.

- `pair<T,U>::create()` Creates empty pair.
- `pair<T,U>::create(ptr< pair<T,U> >)` Creates copy of another pair.
- `pair<T,U>::create(::std::pair<T,U>)` Creates copy of the matching ::std::pair.

Correct usage of pair should be as follows.

```
using namespace ::lestes::std;
typedef map<object,ulint> map_type;
typedef pair<map_type::iterator,bool> pair_type;
// in method body:
ptr<map_type> a_map = map_type::create();
...
ptr<pair_type> result = pair_type::create( a_map.insert(...) );
```

19.1.5 Templated classes

It is vital to obey the restrictions even when creating templated classes. This task is not straightforward. Like for non-template classes, the templated class has to be inherited from collectible base, declare pointers to collectible classes and override the `gc_mark()` method. The problem occurs with the types of the fields specified in the template parameters. It is not known prior to the instantiation, whether the types will be collectible or simple. Thus the proper declaration of fields as `srp` and variables as `ptr` and definition of body of `gc_mark()` method would not be possible.

For this reason, helper entities, declared in lestes/pointer_helpers.hh are introduced. The types defined in `convert<T>::to_ptr` and `convert<T>::to_srp` are used to obtain the respective smart pointer variant from an arbitrary type `T`. The simple types remain unchanged by the `convert<T>` template. These converted types are then used to declare the fields and variables[3]. We also cannot call the `gc_mark()` method uniformly for all fields. If the field `x` is of simple type, the code `x.gc_mark()` would not compile. Instead, a call to a function `gc_mark_ptr(x)` is written for each field `x` with type coming from template argument. The function will itself call the `gc_mark()` method when necessary. Calls to `gc_mark()` for fields of types not dependent on the template parameters are performed the same way as in non-template classes.

19.2 Implementation

The memory manager interface classes reside in lestes/std directory in `::lestes::std::mem` namespace, the implementation part is located in lestes/std/mem directory in `::lestes::std::mem` namespace.

19.2.1 Garbage collector structures

The core part of the implementation is in class `gc`, representing garbage collector. Internally, the garbage collector works with *keystones* and two types of wrapped bare pointers, *simple* and *root* pointers. The task of the garbage collector is to examine the *reachability* of keystones through the pointers, to *mark* the reachable and to *sweep* the unreachable and release them.

19.2.1.1 Keystones

Keystones represent dynamically allocated objects known to the garbage collector. The `keystone` class contains two internal bare pointers. The first links to the next keystone in a linked list of all keystones. It is necessary to be able to access them even when they are not reachable. The second bare pointer links to next marked keystone in the linked list of marked keystones, constructed only during the run of garbage collector. When a keystone is not marked, the pointer is set to `this` to distinguish it from `NULL`, designating the end of the list. Both lists, managed by the `gc` class, are only singly linked, because the elements are only added and never removed before the garbage collector run. Keystones are internally inserted into the marked list by the call to `enqueue()` method. The method `sweep()` performs releasing the keystone, if it is not marked. The `keystone` class is a base for the `object` class, and it should not be used directly in the client programs.

[3] When referring to, for example, `convert<T>::to_ptr` within a template, it has to be prefixed with the `typename` keyword to state that it designate name of a type.

19.2.1.2 Simple pointers

Simple pointers represent bare pointers that are contained in collectible objects. They are not considered when starting the search for reachable keystones, only when the surrounding collectible object is itself reachable. The simple_pointer class contains only the wrapped bare pointer to keystone, its memory representation should ideally occupy no extra storage. It contains a non-virtual gc_mark() method, which checks if the contained pointer to keystone is not NULL and calls its enqueue() method. The simple_pointer is a base of the srp<T> templates, which take care for the proper casting of types.

19.2.1.3 Root pointers

Root pointers represent bare pointers that are not contained in collectible objects, but are otherwise accessible in the program. They include local variables and variables having static storage duration (namespace variables, static fields of classes, static variables in functions and methods). They are the starting points for the reachability search. The root_pointer class, contain links to other root pointers. The root_pointer is the base of the ptr<T> template.

19.2.2 Garbage collector operation

The implementation uses a tracing mark-and-sweep garbage collector. It traces the *live* objects, marks them and sweeps the unmarked objects. An object is recognized as being live, when it is pointed to by any of the root pointers, or by a simple pointers contained in the in collectible objects transitively from a root pointer to the object.

19.2.2.1 The marking method

There are two kinds of gc_mark() method. The first one, defined in simple_pointer, marks the pointee object. The second one, defined in keystone, is virtual and should mark all objects pointed to by the most derived object. When called on a keystone instance, the overridden gc_mark() method contains field.gc_mark() calls for collectible fields of the class of the most derived object. Each gc_mark() calls enqueue() on the pointee keystone, effectively marking such keystone reachable. The ancestor::gc_mark() call ensures recursive marking of fields of all subobjects, so that fields of the entire object are marked and not only the fields added in the most derived object.

19.2.2.2 Mark phase

The garbage collector runs a *breadth first* search. It firstly marks the level zero of keystones reachable via root pointers by calling their enqueue() method. Then it calls the gc_mark() method for the level zero keystones. This way a first level of marked keystones is created, which are reachable via simple pointers from the level zero of keystones. This continues until the constructed levels are nonempty.

19.2.2.3 Sweep phase

When all reachable objects were added to the marked objects, the sweep phase begins. The garbage collector walks through the list of keystones, calling a sweep() method for each. If the keystone is not marked, the sweep() method calls delete this and releases the most derived object, because the keystone destructor is virtual. If the keystone is marked, it means that its object is live. The mark flag is reset to this and the keystone is linked again into the list of all keystones. At the end of the sweep phase, only live collectible objects remain.

19.2.3 STL wrappers

Each wrapped STL container is a templated class with parameters corresponding to parameters of the original containers. The wrapped containers are inherited from the STL templates and object so that they can become collectible classes. The elements of the containers can be simple types or srp<T> pointers, in the case of map container, combination of them. The reason to include wrappers for containers of simple types is that these containers themselves can be allocated dynamically and so require to be handled by the memory manager. The implementation of all containers thus overrides the gc_mark() method, which walks through all the srp<T> elements with iterator and marks each of the objects.

The problem with this design is, that the instantiation of STL containers with elements of type srp<T> declares all methods as manipulating with the simple pointers, which can not be traced. This also applies for iterators

created to point to these containers. However, we can imagine the instance of a container as a black box containing pointers to some collectible objects. By assuring that the instance of the container is reachable from the root pointer set, all the contained objects will also be reachable, because all the stored pointers are always visible through the iterator called in the `gc_mark()` method.

19.2.4 Implementation issues

When designing the memory manager, we tried to use paradigms of object oriented programming. Due to performance issues, almost all methods were declared `inline`, exposing their implementation. Also the `root_pointer` and `keystone` classes were declared `friend` in the `gc` class and vice versa, to be able to access the private linked lists of root pointers and keystones.

Multiple inheritance of collectible classes was intentionally not supported, as its implementation would enforce the utilization of virtual base classes, whose support in compilers is generally weak. However, it is possible to inherit from multiple classes, when it is ensured, that at most one of them is collectible. Managing the memory resources of the other base subobjects is completely in the responsibility of the developer.

Construction of wrapped STL containers from iterators is not supported, as all tested versions of the used compiler generated invalid code for certain cases of the construction calls.

Chapter 20

Message reporting

20.1 Introduction

Diagnostics is an important part of Lestes compiler. It is performed through issuing reports to the user. In the following sections, we describe the client programmer interface and the implementation of the component.

20.2 Reporting Interface

20.2.1 Report structure

Each report consists of a text *message* accompanied by *location* in the source. The main purpose of the reports is to inform the user about problems encountered while processing the source files. Example of two reports:

```
file.cc:13:7: #else without #if
file.cc:1:1: the conditional starts here
```

20.2.2 Location

Diagnostic messages have to be reported within the relevant context, which is supplied in the location information. Each location consists of the name of the source file, line number, column number and further information about position of the source file in the inclusion chain. The location information is passed between the internal compiler structures, so that it is possible to issue a report in arbitrary stage of the processing.

20.2.3 Message

The message is the text part of the report passed to the user. To enable more specific diagnostic, messages can contain parameters. Parameters are positions in the message, into which a piece of arbitrary text information is inserted each time the message is reported.

20.2.3.1 Structure of message

Each message has several properties:

- Type, which is either *warning* or *error*.
- Unique name, which has to be a C++ identifier.
- Fixed number of parameters, ranging from zero to ten[1].
- Message content, which is a special format text string.

[1] Only up to three parameters implemented.

20.2.3.2 Message format string

The *format string* consists of plain text interleaved by *formatting sequences*, which start with percent sign `'%'`. Parameter references are represented as percent sign followed by a single decimal digit `'0' - '9'`, designating the index of the parameter to be used at the position. Parameter numbering starts from zero, so for example `'%0'` refers to the only parameter of a one-parameter message. Using a reference to a parameter out of range for a given message is not allowed. To use the percent character literally in the format string, it must be doubled. Any other character or the end of the format string after percent is not allowed.

20.2.3.3 Message parameters

Each parameter has three properties:

- Position is the number used to referring to the parameter in the format string.
- Data type of the corresponding argument, can be either simple or collectible.
- Formatter describes how to transform the argument into text string.

20.2.3.4 Parameter formatters

The value of the arguments supplied to a message have to be converted to text before being used in the report. This is achieved through *formatters*. There exists a templated class `::lestes::msg::formatter`, which represents the implicit formatter. The implicit formatter uses formatting through standard `ostream&` << operators. In case the implicit formatter does not work with the user parameter type, it is necessary to explicitly specialize the formatter template for the parameter type. If the default formatter is not adequate in some case, it is possible to inherit a new formatter with different behavior.

20.2.4 Defining messages

To separate messages definition from the sources, XML format is used. Parametrised messages are defined in Lestes Message Definition (LMD) XML format, stored in *.lmd source files. It is recommended, if possible, to write a separate LMD for each C++ source file. This way the changes in the message file remain local and do not force rebuilding more sources than necessary.

20.2.4.1 LMD structure

The structure of the LMD format resembles the LSD files and it is also transformed to C++ source files later. The LMD format supports several general tags common to LSD format and several specific tags used for defining the messages. The root element of the whole file is <lmd xmlns="http://lestes.jikos.cz/schemas/lmd">

20.2.4.1.1 General tags The common tags are <file-name />, <dox />, <packages />, <foreign-class />, <imports />, <implementation-imports />. Their meaning is the same as in LSD files, for their description, refer to LSG documentation. In the context of LMD, <foreign-class /> and <imports /> can be used to declare types used in the parameters, while <implementation-imports /> is necessary for declaring user defined formatters. It is recommended to set <packages /> to the same package, in which the message will be used, to avoid unnecessary qualification.

20.2.4.1.2 Message specific tags The message specific tag <error /> is used to define an error message. The tag has **name** attribute and contains <dox />, <text /> and <param /> tags. The <dox /> tag is used to document the message. It contains <bri /> and <det /> tags for brief and detailed documentation of the message respectively. The <text /> tag contains the format text for the message as described previously. Each <param /> tag defines one parameter. The attribute type specifies the data type of the parameter. The collectible types are not recognized, so it is necessary to write the entire expression 'ptr < class_name >'. The optional parameter formatter can specify a name of a special formatter class for the parameter. The order of <param /> tags determines the order of the parameters of the message.

20.2.4.1.3 Example message definition

```
<error name="message_name">
   <dox>
      <bri>Brief documentation.</bri>
      <det>Detailed documentation</det>
   </dox>
   <text>the format text with references to parameters: %0 %1</text>
   <param type="simple_type" />
   <param type="ptr &lt; collectible_type &gt;" />
</error>
```

For defining warnings, <warning /> tag is used instead. Its internal structure is equal to that of <error />.

20.2.5 Reporting messages

To report a specific message, it is necessary to `#include` the .m.hh header file containing declaration of the message. This header file, generated from the corresponding .lmd file, also contains all other declarations necessary for reporting messages. The reporting itself is done by using << operator to send the message, followed by all arguments and the location into report object. Example:

```
report << message_name << argument0 << argument1 << location;
```

When the message was defined in another namespace, it is necessary to qualify the name of the message. To properly issue a report, all arguments as well as the location have to be sent into the object.

20.2.6 Building

For the .lmd files to work, it is necessary to add them into `SFILES` variable of the local Makefile. The object files .m.o have to be added to the `exec_name.object` variable containing objects to be linked into the executable named exec_name.

20.3 Reporting Implementation

20.3.1 Introduction

The design of the reporting framework was trying to fulfill several requirements.

- Make defining and using the messages easy for the developers.
- Separate the definition of messages from the source files.
- Keep the message definitions local to the place of their use.

We chose to store the messages in separate files in XML format, which solved the requirements.

- The chosen format is simple and easy to use, being similar to the LSD format used in the project.
- The messages are not hardcoded in the source files, but are defined elsewhere.
- Each module of the project has its own message definition file, thus modification will only affect the parts using it.

20.3.2 Structure

The implementation of message reporting consists of several cooperating parts: the C++ classes for messages, reporting and support and the XSL templates for transforming the LMD source files into C++ source files. The C++ part of the implementation is located in `lestes/msg` directory in `::lestes::msg` namespace, the LMD part is in `util/lsg` directory.

20.3.3 Location

Location describes the position of an entity in the translation unit. Because the translation unit is a result of preprocessing, it is not sufficient to express the position within one source file. Inclusion of other source files has to be also considered, so that the location reflects the forming of the translation unit. Location is represented by classes `source_location` and `file_info`. Class `file_info` contains file information: the source file name and location of the file in the inclusion chain, `NULL` for top-level file. Class `source_location` contains the line and column number and file information.

20.3.4 Message oriented classes

20.3.4.1 Message stencils

Parametrised messages are represented by the `message_stencil`*X* templates, where *X* is the number of parameters for the message. All templates are inherited from `message_stencil`, which holds the common attributes. Each template is parametrised by the data types of the message parameters, which can be of simple or collectible types. The stencil with no parameters has a fake parameter to make it a template as well.

The instance of the stencil is initialized with flags (error or warning), format string and formatters for all parameters. Checking is done on the format string to catch errorneouserroneous constructs (wrong parameter references). Internally, each instance holds a unique identification number, which can help tracking the origin of messages. By calling the `format()` method and supplying arguments, a concrete formatted message is created.

20.3.4.2 Messages

Concrete messages are represented by the `message` class. The message instance holds the flags, the formatted message text and identification number of its stencil. Although `message` contains all necessary information, it is not intended to be created or manipulated directly by the programmer, it is only internally passed to the reporter.

20.3.5 Reporting oriented classes

20.3.5.1 Reporter

The class, which is in charge of issuing reports, is the `reporter` singleton. The report itself is not represented by any object, it is only a common name for the message and location tuple. The reporter has `filter` field, pointing to the start of a chain of report filters. By calling `report(ptr<message>, ptr<source_location>)` method on the reporter, the report is passed through this filter chain.

20.3.5.2 Report filters

The filters are responsible for presenting the report to the user according to the specified requirements.

Each report filter has to be inherited from abstract `report_filter` class. Each filter is chained to the next one through the output filter link. Filters have `process(ptr<message>, ptr<source_location>)` method, which performs the action of the filter. Each filter shall either discard the report or pass it (possibly after modifying it) to the next filter by calling the inherited `process_output(ptr<message>, ptr<source_location>)` method. There are several filters, each with its own purpose.

- `report_kind_filter` - Discards reports by message kind, used to suppress warnings.
- `report_error_flag` - Watches for first error message.
- `report_origin_filter` - Filters origin, to avoid displaying the inclusion chain repeatedly.
- `report_cache` - Saves last processed report, for debugging.
- `report_ostream` - Writes the report to ostream, the actual displaying of the report.
- `report_end` - Discards all reports, used to terminate the chain, because `NULL` report is not allowed .

20.3.6 Supporting entities

The possibility of reporting easily via the `<<` operator, it is necessary to introduce support macro, classes and operators. When issuing a report by writing

```
report << message_name << arg_zero << arg_one << location;
```

a lot of various operation has to be performed to achieve the result. The identifier `report` itself is only a macro expanding to `::lestes::msg::reporter::instance()`. Each of the operators is overloaded to store the argument into the parameter holder.

20.3.6.1 Argument holders

The message stencil and the arguments used for creation of the messages are saved in `argument_holderXY` template family. The *Y* digit designates how many arguments are already saved, the *X* digit is equal to the total number of parameters of the message stencil. The template parameters are the types of the saved message arguments. The `argument_holderX0` is initialized with the stencil, `argument_holderXY` with the previous holder `argument_holderXZ`, where *Z = X - 1,* and the corresponding argument.

20.3.6.2 Overloaded operators

The first overloaded `operator<<(ptr<reporter>, ptr<message_stencilX>)` is used to save the stencil and returns `argument_holderX0`. The rest of operators takes an argument holder and argument and returns the next argument holder. The last `operator<<(ptr<argument_holderXX>, ptr<source_location>)` collects the stencil and parameters from the holder, formats the message and sends it into the reporter.

20.3.7 LMD transformation

The LMD files are transformed with XSL templates, **lmd2hh** and **lmd2cc**, creating .m.hh and .m.cc files, respectively.

20.3.7.1 Mapping of general tags

The tags common with LSD are used to create correct C++ source file structure. The result of transforming these tags is the same as in LSD. The <dox /> tag becomes file Doxygen documentation. The <package /> tag is changed into `package()` and `end_package()` lists. The <imports /> and <implementation-imports /> tags become `#include` in the .m.hh or .m.cc file respectively. The <foreign-class /> tag turns into forward declaration of the class.

20.3.7.2 Mapping of message tags

Each <error /> or <warning /> tag defining a parametrised message with X parameters is transformed into static storage duration variable of that name defined in the namespace specified by the <package /> tag. The type of the variable is `ptr` to `message_stencilX`, the template parameters are the contents of **type** attribute <param /> tags, if any. For `message_stencil0`, fake `bool` parameter is supplied. The defined variable is initialized with flags (`FLG_WARNING` or `FLG_ERROR`), the content of the <text /> tag converted to string, and the formatters for all parameters. When the **formatter** attribute is missing for some parameter, the default `formatter<T>::instance()` is used instead.

The .m.hh header file contains merely the declarations of the variables, which is enough for using the message. The .m.cc file contains the definition and initialization of the variables. This is also the reason, why it is sufficient to mention the sources declaring the user defined formatters in <implementation-imports />, because they are used only in the initialization.

20.3.8 Support in build system

The central Makefile handles .lmd files listed in the SFILES variable of local Makefiles, like it handle .lsd files. In the gen phase, .m.hh and .m.cc files are generated from .lmd files. The .m.cc C++ sources are then compiled as regular source files.

20.3.9 Further improvements

The XML structure can be extended to support localization, by adding the attribute **language** to the <text /> tag and modifying the XSL templates appropriately. This way, all localized variants of each message would be stored in its definition. The order of message parameters can also be changed appropriately to reflect the syntax of the language.

Part V

Executing Lestes

Chapter 21

Compiling Lestes

21.1 System Requirements

To compile the Lestes project, the following supporting utilities have to be installed on the system where compilation will be done

- **GCC** compiler. We strongly recommend gcc version **3.4.3**. We have seen serious bugs in code generated by gcc 3.3.x (and sometimes gcc 3.2.x) when compiling our code. Compilation with gcc4 was not even tried.
- **GNU bison** parser generator. We only support versions **1.875** to **1.875d.** Other versions are not supported.
- **GNU flex** lexical analyser generator. Version **2.5.4** or higher should be used, we have experienced problems with older versions.
- **libxslt** library (together with **xsltproc** processor). At least version **1.1.11** is needed. The older versions contain a bug which causes failures when compiling the Lestes compiler.
- **libxml2** library of version **2.6.15** or higher.
- **GNU make** of version **3.80** or higher. It is possible that older versions also work, but we haven't tested them.
- **makedepend** dependencies generator. This utility is part (maybe quite surprisingly) of X11 distribution.
- **sed** stream editor.

To compile and link programs using the Lestes compiler, the following software is needed

- **lestes** binary
- **nasm** assembler compiler. Any recent version should work flawlessly.
- **GNU ld** linker. Any recent version should work flawlessly.

When Lestes compiler is instructed to dump intercode structures into XML dump, the transformation to the HTML format is usually requested by user, to allow better orientation in quite complex interconnected structures. Also, the internal loggers (when they are switched on) of Lestes compiler, parse the configuration files (if any) written in XML format. For these two tasks, the following packages are required:

- **libxml2** library of version **2.6.15** or higher
- **xsltproc** or any other XSLT processor [1]

[1] This is needed just in the cases when user has asked the Lestes compiler to dump the intercode structures, and is willing to use the prepared XSLT template to transform them to HTML.

21.2 Compiling Lestes

If all the prerequisites specified in the previous section are met, the Lestes compiler can be compiled by simply invoking `make`, without any arguments, in the top-level project directory. The result of this compilation is `lestes` binary, which is placed in the `runner/` directory. The further usage of this binary is described in the following chapter.

It can happen that user trying to compile Lestes compiler receives the following error message during compilation

```
bison: cannot open file '/usr/share/bison/m4sugar/m4sugar.m4': No such file or directory
```

This happens due to wrong installation of the bison on the compilation host. The quick fix to workaround this problem is to download the bison source codes from GNU webpage `http://www.gnu.org/software/bison/`, unpack them to the home directory of the user, and set environment variable `BISON_PKGDATADIR` to the directory `data/` in the unpacked directory. For example, when the archive is unpacked to the directory `~/lestes/bison-1.875/`, the command that has to be issued prior to the compilation would be

```
export BISON_PKGDATADIR=~/lestes/bison-1.875/data
```

Chapter 22

Executing Lestes binary

Upon a successful compilation and linking of the project tree, the main binary used to invoke the compiler, is located in the `runner/` directory and is called `lestes`. When run without any arguments, it prints a short usage help. The binary accepts the following command line options:

- `-I directory` - this option has the same meaning as it has in gcc compiler (and others). It adds the specified path into the list of directories that are searched through when `#include` directive is used in the source. This means that if the programmer has a file `foo.h` in `/foo/bar` directory and desires to be able to write

  ```
  #include <foo.h>
  ```

  in his or her source code, specifying the `/foo/bar` path using the `-I` parameter is needed.

- `--as filename` (also `-a filename`) option instructs the compiler to dump Abstract Syntax structures into the file called `filename`. Structures are dumped into the file as soon as the parser succeeds parsing the input source. This is very useful when one wants to inspect the structures as they are directly after the parsing phase has completed. This option can be also used for ensuring that structures produced by parser are correct, even if the compiled source is not able to pass all the compilation phases (due to either semantic error, or unimplemented feature in other parts of Lestes compiler).

- `--ss filename` (also `-s filename`) option instructs the compiler to dump Semantic Structures into the file called `filename`. Structures are dumped into the file as soon as the semantic analysis part has been completed and the semantic tables are prepared for further processing by ss2pi and backend phases. This option can be used for ensuring that structures produced by parser and SA parts are correct, even if the compiled source is not able to pass all the compilation phases (due to either semantic error, or unimplemented feature in other parts of Lestes compiler).

- `--stop-after-ss` options instructs the compiler to stop processing the structures after the SS structures (i.e. intercode tables) have been filled. This is useful when inspecting a feature of the compiler that is implemented by frontend but not by backend. This allows user, in combination with `-ss` option described above, to view the intercode structures even in such cases. **Note:** this can be also achieved automatically for given source, when string `#pragma lestes` is inserted anywhere into the source.

- `--ctx filename` (also `-c filename`) option instructs the compiler to dump the SA context into the file called `filename`. This is also performed in the situation that assertion fails in internals of Lestes compiler.

- `--llc filename` (also `-l filename`) option instructs the compiler to load logger configuration from the file called `filename`. This configuration file should contain settings for the logger tree of the Lestes compiler, allowing to turn on/off the specific loggers. This is used mainly for debugging purposes.

- `--llc-skeleton filename` option instructs the compiler to dump the skeleton of the logger tree into the file called `filename`. The resulting skeleton can the be manually edited to toggle specific loggers on/off and the run the compiler with this modified file, using the `--llc` option mentioned above.

- `--output filename` (also `-o filename`) option specifies the filename to which the resulting assembler output should be written. If omitted, standard output is used.

- `--deep-dump` option instruct the dumper part of the Lestes compiler to make all the dumps "deep", i.e. human-readable.
- `--stdin` (also `-`) option instructs the compiler to read the source code from standard input.
- `--help` (also `-h`) option prints usage help and exits.

After the options are specified on the command line, the input filename, containing the source to be compiled (or `--stdin` switch mentioned above) has to be specified.

22.1 Compiling the assembler output. Linking the executable.

The output of Lestes compiler is assembler code, which can then be used as input to **nasm** assembler, producing ELF object file. This ELF object file has to be the linked against system libraries, to produce executable ELF binary.

Let's assume that we are trying to compile source `overload.cc` source in the `runner/examples/` directory, which is also the current directory. The commands to produce executable `overload` would then be

```
../lestes -I ../../lib/ -o overload.asm overload.cc
nasm -g -f elf -o overload.o overload.asm
g++ -o overload overload.o ../../lib/lio.o
```

The first command invokes the Lestes compiler itself and produces the assembler code. The second command invokes `nasm` assembler to produce ELF object file. The last one calls the system linker (which is invoked using g++ frontend) to link the object file `ab.o` with the system objects needed to produce executable binary (`crt1.o` and others).

The `lio.o` library mentioned in the example above is discussed in the following chapter.

Chapter 23

Example source codes

23.1 LIO - Lestes IO library

Lestes provides it's own trivial library for the purposes of performing I/O operations. There are two main reasons for providing such library - the Lestes compiler in it's current state doesn't support functions with ellipsis arguments, so we had to create wrappers for commonly-used combinations of calls to `printf()`. The correct one to be called is then chosen by overload resolution, depending on the types of passed arguments.

The second reason is that there is currently no way how to instruct the compiler to use C name mangling, so the library itself has to be compiled using standard C compiler and the linked with objects originating from Lestes compiler (this also, as a side effect, demonstrates that our compiler produces code that can be linked with library produced by GCC compiler). The sources in `runner/examples/` have to include `lib/lio.hh` and the linked with `lib/lio.o`.

23.2 Example Sources

There are examples, located in `runner/examples/` directory, demonstrating the individual features of the compiler and the language constructs. Each of them contains at it's very beginning the short description of the feature it is demonstrating and also explains how to use the example. Maybe the most elaborate one, demonstrating pointer arithmetics, preprocessor macros, while statements, if statements, overload resolution and function calls, pointer dereferencing, etc. is the source called `big_factorial.cc`, which is computing factorials for large number, using string operations.

If the user asks the compiler to dump the intercode structures, they are dumped into the XML file. The XSLT template for transforming the dumped XML file into HTML, which can be easily viewed and navigated through by any web browser, has been written. If the user wants to transform XML structure dump into it's HTML equivalent, the template is applied in the following way:

```
xsltproc ../../util/dump-vis/vfd.xslt dump.xml > dump.html
```

(assuming that the current working directory is `runner/`, otherwise the path to the XSLT template should be adjusted accordingly).

There exists the `fail` subdirectory in the `runner/examples/` directory. This directory contains examples of sources, purpose of which is to demonstrate that errors in source codes are correctly detected and reported. Each of them contains in it's comment at the beginning of the file description of the problem it is demonstrating.

Both of the directories `runner/examples/` and `runner/examples/fail` contain `Makefile`, used to build sources in them. If invoked without parameters, all available sources (or, to be more precise, all targets specified in the `TESTS` variable inside the `Makefile)` are built. It is also possible to specify only one target to be built. Let's assume that we want to build `runner/examples/overload.cc` into the executable `overload` binary. This is achieved by running command `make overload` in the `runner/examples/` directory. This can be particularly useful in the `fail/` subdirectory, as the compilation fails for sources in this directory, so `make` builds only the first one it hits, so the others have to be specified manually, if desired.

Part VI

Conclusion

Chapter 24

Project History and Design Decisions

24.1 Analysis

In the planning phase of the project, we had to study both literature on compiler construction and – more importantly – the Standard of the C++ language.

The Standard is not only quite large, it is also very complicated to comprehend during the first couple of readings. We spent more than a year trying to understand what exactly is needed to be implemented and how. This part of the analysis was very important because all the features of the language are so interconnected that an overlooked characteristic or construct would probably cause a complete overhaul later, when the deficiency is found.

During the initial analysis of the task the team has divided into groups that were working on different issues of the design. These groups were covering the following parts of the compiler: preprocessor, parser, semantics, architecture-dependent actions. Others focused on investigating the underlying development framework: intermediate representation, automated generation of its structure, and design patterns utilization.

24.2 Development Framework

24.2.1 Structure Generator

Basic support for data structures in Lestes project is provided by the Standard Template Library. However, in the early days of designing we conceived that we need more than this. First of all, C++ lacks automatic garbage collection, which is necessary for such a large project. Likewise, we wanted automatic support for dumping the structures at any time for the purposes of development and debugging (and educational purposes in the future). Later it turned out that support for specific design patterns would be quite useful. We have chosen XML as a suitable tool for description of the structures, since it is a standardized meta-language for structured documents and it is flexible enough. The XSLT language emerged naturally as a tool for automatic processing the structure description and generating C++ classes with the desired support. The total number of classes in the project exceeds one thousand and it would be impossible to maintain it fully by hand.

24.2.2 Design patterns

Bringing some of the design patterns to practice turned out to be quite painful. In the end they proved to be inevitable, learning to think differently was not easy however. As we did not have much experience with these, we were not sure what to expect and what to require. Very good example of misunderstanding were the visitors and their integration within the generator: support for them had to be reworked several times until an acceptable one was introduced. Nevertheless, the final version is robust enough to provide us with automatically-generated code without which the project would not be possible.

24.2.3 Memory management

In an early stage of the design, it became clear that manual deallocations of dynamically allocated structures would be nearly impossible. Thus the concept of automated memory management was introduced. After several attempts, we implemented a lightweight garbage collector, suitable for our purposes.

24.2.4 Build system

The build system had undergone quite a rapid changes during the different phases of the project. because in the beginning it was not clear what features will be needed and what features are useless for us. Our project is heavily using automatically generated code for various purposes. There are XML files, describing not only data structures (which was the original idea), but also algorithms, design patterns (visitors), error messages, configuration for debugging output, etc. Specific groups of these files have to be handled in slightly different ways.

We immediately hit problems with dependencies. As we are mixing manually written source code and automatically generated sources, generating dependency graph is not a straightforward task.

In the current project stage, we ended up with four-phase compilation process. During the first phase, the whole source tree is recursively processed, and dependencies between our XML files are generated (we had to write our own XSLT template, which will process XML files and emit dependencies in the GNU Make compatible format). During the second recursive phase, XML files are processed by XSLT processor and our templates, and C++ source code is generated. The third phase is again a dependency generating phase - it is not possible to generate dependencies between C++ sources sooner, as the generated sources were not up-to-date or did not exist at all. The fourth phase compiles all C++ sources into corresponding object files, and the last phase links object files together. The fourth and fifth phases could not be merged into one, as the linking is cross-directory, i.e. one resulting file may be product of different object files from different subtrees, and dependencies between these files can't be easily described when running build system recursively.

Our project also desires extensive testing and running individual tests every time after compiling the source tree was not acceptable. However, different test cases have different needs (definition of what "success" means is not clear and uniform - for one test successful run means that parser accepted the source code but semantic analysis rejected, other test's correct behavior is to make parser fail, another one must compile flawlessly and give predefined output, etc) so the build system has to be prepared for this.

24.3 Compiler

24.3.1 Lexical analysis (preprocessor)

Unlike in other programming languages (Pascal, C#, Java), preprocessing of C++, described in the ISO Standard, is an important part of C++ lexical analysis and cannot be omitted. We considered three major approaches to implementing the preprocessor.

24.3.1.1 External program

The first approach was to omit implementing the preprocessor entirely in favor of using an external preprocessor. Before being analysed by the compiler, the source code would be sent to a standalone preprocessor. The preprocessor would carry out all the macro expansions and inclusions itself. The resulting preprocessed translation unit would be only lexically analysed to obtain the tokens.

Although this choice is suitable for an experimental compiler, the idea was abandoned. The reason is because the available freestanding preprocessor, GNU cpp, did not implement certain requirements of the C++ Standard (universal characters), while it was extended by many unwanted features. Also tracking the location of errors in the preprocessed source would be quite difficult.

24.3.1.2 Wrapped module

The second approach was to encapsulate an open source third party preprocessor into the compilers sources and to write wrapper code to interface with the preprocessor. Handling of the input files would be managed by the compiler, which would pass their content into the preprocessor. The compiler would then pick up the tokens and continue their processing. The candidate was the Wave C++ preprocessor, a part of Boost library open source project, dedicated to programming using advanced C++ techniques.

This approach was abandoned for several reasons. Firstly, the Wave preprocessor was in a preliminary version and the support for the preprocessor features was questionable. Secondly, it was extensively using the non-standard Boost library, which could endanger the compilability and portability of the compiler source code. Moreover, the error reports coming from the preprocessor could be insufficient for the purposes of the compiler and would be incompatible with the designed error reporting support.

24.3.1.3 Own implementation

The third approach was to implement the preprocessor from a scratch. The main advantage of this choice is simple interfacing with the rest of the compiler, chiefly the parser and the error message reporter. Another advantage is implementing and supporting exactly the features required by the ISO standard, with the possibility to choose only a subset of features, vital for operation of the rest of the compiler. We chose to write our own implementation, which is suitable for our purposes.

24.3.2 Syntactic analysis (parser)

We succeeded at implementing a parser that deals with peculiarities of the C++ language quite well. After a few attempts failed, bison-generated LALR(1) parser suite was chosen. What makes our solution special is that ambiguity resolution is performed while the main parser is paused. We try the alternatives possible by the grammar one after another, until one that actually matches the input is found. Implementing this in bison was not trivial at all. The most problematic is the property of the finite state automaton it is based on, namely the look-ahead token. Great care must be taken to run the sub-parsers before the automaton looks ahead and makes a decision based on the incoming token value. Therefore disambiguation is in almost every case started *after* a construct that has been processed rather than before a possibly ambiguous one, as the latter might involve reading the look-ahead token.

24.3.2.1 GLR

The idea of trying to parse the input and see what the input can actually mean is similar to the GLR parsers. They approach the problem by running the parsers in parallel. Whenever there are more possibilities to choose from, GLR parser *forks* to be able to cover all possibilities. The parsers then work synchronously, which means that they all read the same input token and they do it at the same time. Whenever a syntax error occurs, the parser in question simply passes away. When no parser is left, the game is over, there really was an error on the input.

Although writing these parsers is much easier, compared to changing the grammar to allow for disambiguation breaks, one has much less control over the parsing process. As all the parsers are equivalent, it is difficult to find out whether the current branch is the one that will be confirmed or it will come to an dead end. Lack of control was the main reason we stepped away from GLR and chose the LALR(1) automaton.

24.3.2.2 Super-set of the language

Another failed attempts include single LALR(1) parser that would not itself solve any of the ambiguities of the language, but rather leave this task up to the semantic analysis. The grammar would have to cover a super-set of the C++ language. Presumably, the super-set grammar would not resemble the one in the standard.

This path was abandoned because the disambiguation would be far too difficult to accomplish in semantic analysis. Predecessor of the final parser was a concept that assumed that the correct path can be recognized by trying to match the ambiguous input against regular expressions. For relatively long time this was being worked on until we realized that the specially-crafted regular expression can be better specified using grammar. That is what the syntactic analyzer does now – rather than matching against special regular expressions (that would have to support nesting, for example), special – yet normally generated – LALR(1) parsers are run.

24.3.2.3 Hinter

With regard to the hinter, it was clear from the beginning that we would need the type information to be passed back from the semantics. All identifiers are tagged with type information before they enter the parser. It would be possible to only do the search when really needed, but this was deemed not worth the effort. The hinter would have to be turned on and off from the parser depending on the context, which would clutter the grammar rules.

24.3.3 Semantic analysis

We modeled our intercode on the WHIRL (see the documentation `http://www.cs.ualberta.ca/%7Epengzhao/ORC/whirl.pdf`) at first. We took the layered philosophy of WHIRL, yet we abandoned the actual representation of structures, because they were poorly suited to the objective design paradigm. The five layers of WHIRL were subsequently subdivided and merged to form seven-layer architecture. After more detailed expression transformation analysis, it was found necessary to further subdivide the layers to form a total of nine layers. Shortly afterwards it was however found out that the separation that was the basis for differentiating various levels is impossible, because C++ language mandates complete analysis of subexpressions before the whole expression

transformation starts (e.g. in order to determine the type of a subexpression it is necessary to perform overload resolution on that subexpression).

The next requirement of the project assignment was to use DAG-like form of intercode. We explored the possibility to use SSA (Static Single Assignment Form) as it is held as the most progressive intercode form used today. We abandoned this approach because of its complexity. The current intercode shares some of the features with SSA (e.g. there is only one write to any given pseudoregister). The cunning encoding of data control dependencies by means of sequence points and origins bring the full power of DAG to our intercode.

24.3.4 Code generation

24.3.4.1 Machine description

Early idea, we had, is that some of existing architecture description languages is to be used. For example LISA, CMDL. Unfortunately, creation of an exhaustive description in such a language is not realistic. On the other hand, descriptions for few processors can be found for each of these languages.

One problem is to find or create a description. Next, and perhaps even bigger problem, is to extract the needed information from it. The language usually uses its proprietary data format and tools that generate C++ sources with classes representing target architecture are very rare and not usable for our goal. Even if one exists, it can be impossible to integrate it into our project.

Contrary way is to create C++ classes for architecture by hand. This has several disadvantages:

- It is not retargetable in any way.
- It is error-prone because of a lot of similar entities (e.g. instructions).
- It is not comprehensible.
- It can become very complex.

The XML format, which is simple and extensible, was chosen for representing machine description. C++ classes are generated from it by XSLT. XML format is favourable because it is standard, there are many tools (such as editors, validators, parsers, processors), we have the know-how thanks to structure generator (LSD) and XSLT templates are simple but mighty tool for transformation to the desired form.

24.3.4.2 Output Format

We had to decide on what output format is to be generated. In early stages of the design, we intended to generate binary object file. By reason of special handling of templates it would be necessary to implement own linker. It also had to be compatible with .obj files generated by GCC in order to be able link with libstdc++. For portability, it is needed to support few object file formats or portable one (DWARF). It seems very complicated and due to lack of time it was abandoned.

We have chosen simple but flexible solution - the backend generates only the assembly file. It has few advantages:

- Description of the asm language can be added to machine description so backend is not asm language dependent.
- It is readable by human. This is important for debugging.
- Information about templates could be passed through asm to object file transparently.

24.3.4.3 ABI

The ABI (application binary interface) specification document imposes many stringent requirements that are not easy to implement. This includes exceptions, VMT, guard variables, RTTI, and DSO object destruction. Additionally, the majority of literature is accessible in the form of drafts only, and is incomplete. These features can be thought of as modules, that can be implemented later. Now only limited subset of features is implemented, that permits us to link against GCC's standard library.

Chapter 25

Conclusion and Future Work

25.1 Conclusion

Quite a indispensable amount of work on the project was initially spent on reading the Standard and preparing a framework, that would be robust enough to ease such a complex task, as writing the compiler for a language like C++ is. This was successfully achieved – as soon as the framework was developed and ready-to-use, the evolution of the compiler itself to it's current shape was quite rapid. This was caused by the fact that the work was properly scheduled in advance when studying the Standard and also because the framework we developed provided us with features which were rich enough to make the work comfortable.

From the very beginning of the project it was obvious (and it directly follows from the project assignment), that the result is not going to be fully-featured compiler. The project was meant to be a good foundation for subsequent development. The fact that the created framework is well-suited for compiler writing task was proved by writing our working compiler, which is modular and open for easy adding of new features. In addition to that, the C++ Standard was deeply and thoroughly studied and according to the gained knowledge, the internal structures for the code representation were designed, implemented and documented.

25.1.1 Doxygen documentation

As this project is meant to be used by other programmers to continue on developing it, the proper commenting of the code and structures was not forgotten. It is supported by our framework, the generated sources have Doxygen-compatible comments and the Doxygen documentation is provided together with the project. As the project currently has over 1000 classes, it is very important that structure, represented by namespace hierarchies, can be easily seen in the resulting Doxygen documentation, allowing it to be quite easy to navigate through and understand the relations between individual classes.

25.1.2 Internal notes

Last but not least, the result outcome of the study of the C++ standard resulted in notes we have created for our purposes. These "caveats" can be used in further development and enhancement of the compiler, to provide information about non-trivial features and properties of the C++ language, that have to be carried on mind and must not be forgotten. These notes are provided together with the compiler.

25.2 Future Work

We have implemented only a subset of the C++ language, specified by the ISO standard. The continuation of our work will involve adding support for these missing features into the compiler. As shown before, a good hinterland for adding such features has been created. The most notable features of the language that have to be added are

- support for constructors in the terms of cooperation between syntactic and semantic analysis
- better support for object types in SA transformations, including member access, indirect function calls, casts. Improving support for pointers to non-builtin types and implementing missing conversions.
- support for object types in the backend part of the compiler

- support for templates
- adding optimizations, either on the intercode level, and also in the backend

It is important to note here, that there are already assigned master theses on top of this compiler, which will continue with developing this code. The assigned theses concern the following parts of the compiler

- parser
- garbage collector for internal compiler structures
- framework - code and structure generator
- backend

Part VII

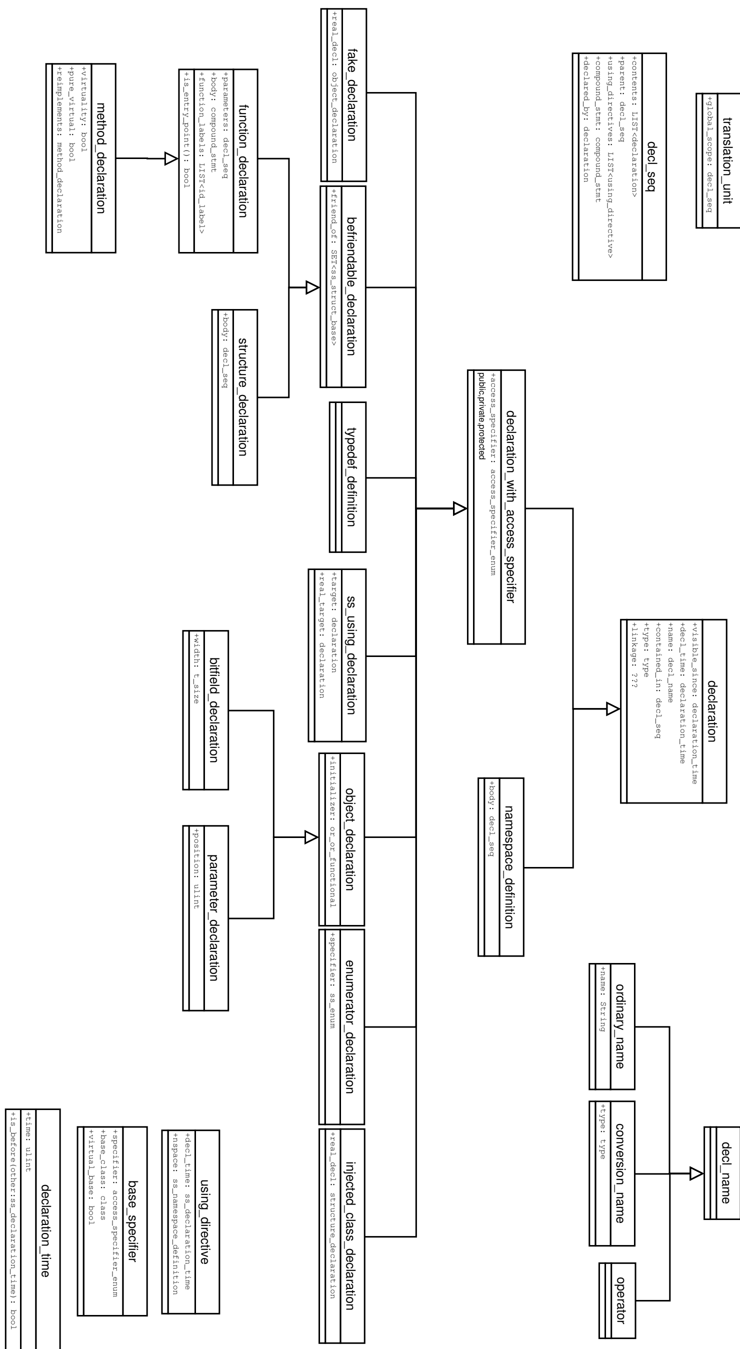
Appendices

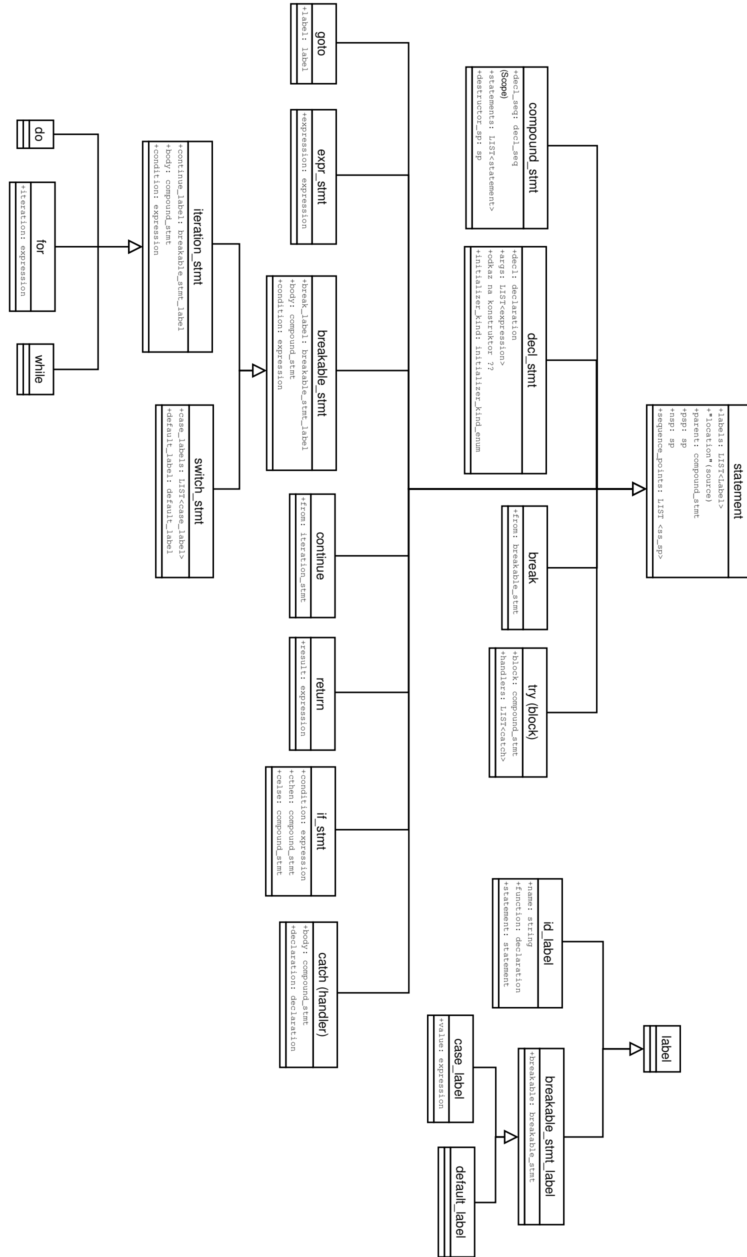
Appendix A

Hierarchy of semantic structures (layer ss)

On the following pictures is a basic overview of the semantical structures on ss intercode layer.

Note, that the naming of structures is without common *ss_* prefix.


method_declaration
function_declaration
structure_declaration
fake_declaration
befriendable_declaration
typedef_definition
ss_using_declaration
object_declaration
enumerator_declaration
injected_class_declaration
bitfield_declaraton
parameter_declaraton
declaration_with_access_specifier
namespace_definition
declaration
decl_seq
translation_unit
ordinary_name
conversion_name
operator
decl_name
declaration_time
base_specifier
using_directive


statement
+labels: LIST<Label>
+"location"(source)
+parent: compound_stmt
+psp: sp
+nsp: sp
+sequence_points: LIST <ss_sp>
compound_stmt
+decl_seq: decl_seq
(Scope)
+statements: LIST<statement>
+destructor_sp: sp
decl_stmt
+decl: declaration
+args: LIST<expression>
+odkaz na konstruktor ??
+initializer_kind: initializer_kind_enum
break
+from: breakable_stmt
try (block)
+block: compound_stmt
+handlers: LIST<catch>
goto
+label: label
expr_stmt
+expression: expression
breakable_stmt
+break_label: breakable_stmt_label
+body: compound_stmt
+condition: expression
continue
+from: iteration_stmt
return
+result: expression
if_stmt
+condition: expression
+cthen: compound_stmt
+celse: compound_stmt
catch (handler)
+body: compound_stmt
+declaration: declaration
iteration_stmt
+continue_label: breakable_stmt_label
+body: compound_stmt
+condition: expression
switch_stmt
+case_labels: LIST<case_label>
+default_label: default_label
do
for
+iteration: expression
while
label
id_label
+name: string
+function: declaration
+statement: statement
breakable_stmt_label
+breakable: breakable_stmt
case_label
+value: expression
default_label

expression
+type: type
+psp: sp
+nsp: sp
+expr_time: declaration_time
+<<virtual>> is_lvalue(): bool
return false
+<<virtual>> get_value(): ss_base
For const expression only

var_ref_abstract
+is_lvalue(): bool
return true

literal
+value: literal_info

ternary(? :)
+condition: expression
+true_expr: expression
+false_expr: expression

unary_expr
+expr: expression

var_ref
+var_decl: declaration
the variable declaration

var_ref_by_qname
+contained_in: decl_seq
+decl_time: declaration_time
+qualified_name: as_name

funcall_abstr
+args: LIST<expression>

get

address_of (&)
+is_lvalue(): bool
return true

dereference (*)
+is_lvalue(): bool
return true

conversion
+src_type: type

bind_reference
+src_type: type

bind_to_temporary
+src_type: type

vol_get

pfuncall
member pointer call
+fun_ptr: expression
+object: expression

funcall
ordinary global
+function: function_declaration

ifuncall
indirect ordinary global
+fun_ptr: expression

mfuncall
method call
+function: function_declaration
+object: expression
??? pointer insufficies, temporaries causa

vfuncall
virtual method call
+function: function_declaration
+object_ptr: expression

binary_expr
+left: expression
+right: expression
+is_lvalue(): bool
TODO return false ?(array)

ptr_access (->)
+object_ptr: expression
+member: declaration

access (.)
+object: expression
+member: declaration

ptr_member_ptr(->*)
+object_ptr: expression
+member_ptr: expression

member_ptr(.*)
+object: expression
+member_ptr: expression

throw
see throw-expr, chap.15
+type: type = void
+value: expression

sp
+psp: sp
+nsp: sp
+nse: LIST <se>
+pse: LIST <se>
+pi_spe: pi_sp
pi ekvivalent,spec.get
+level: ulint

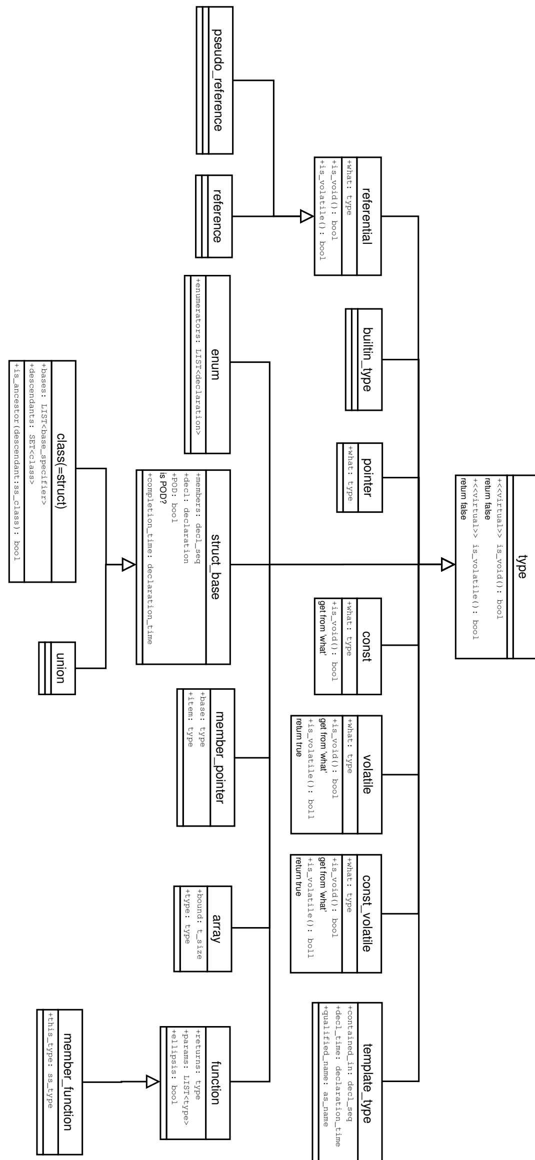
se
+from: expression
+psp: sp
+nsp: sp

assign (=)
+is_lvalue(): bool
return true

[ ]
Transform to pointer ??

+
addition

...


type
+<<virtual>> is_void(): bool
return false
+<<virtual>> is_volatile(): bool
return false
referential
+what: type
+is_void(): bool
+is_volatile(): bool
builtin_type
pointer
+what: type
const
+what: type
+is_void(): bool
get from 'what'
volatile
+what: type
+is_void(): bool
get from 'what'
+is_volatile(): boll
return true
const_volatile
+what: type
+is_void(): bool
get from 'what'
+is_volatile(): boll
return true
template_type
+contained_in: decl_seq
+decl_time: declaration_time
+qualified_name: as_name
pseudo_reference
reference
enum
+enumerators: LIST<declaration>
struct_base
+members: decl_seq
+decl: declaration
+POD: bool
is POD?
+completion_time: declaration_time
member_pointer
+base: type
+item: type
array
+bound: t_size
+type: type
function
+returns: type
+params: LIST<type>
+ellipsis: bool
class(=struct)
+bases: LIST<base_specifier>
+descendants: SET<class>
+is_ancestor(descendant:ss_class): bool
union
member_function
+this_type: ss_type

Appendix B

Special Considerations in Semantic Analysis

This chapter contains a brief summary of the semantic requirements imposed on the source program by the Standard. It also contains the rationale for various decisions made during the implementation process.

B.1 Semantic Checks

1. Insert class-name in class scope **[3.4/2]** (as member); **[9/2]**
2. x
3. x
4. x
5. x
6. x
7. **[9.3/2]** it is not possible to redeclare member functions (except for: out-of-class definitions, explicit specializations of member functions, templated classes, explicit specializations (of templated member functions)). **[9.3/3]** external linkage of MF in namespace scope.
8. inline functions:
   (a) **[3.2/3];[3.2/5]** - defined in every translation unit is is used
   (b) **[7.1.2/3]; [9.3/3]** - there cannot be an `inline` specifier by both the member function declaration and definition. The `inline` specifier cannot be used on a block scope.
   (c) **[7.1.2/4]** - the same definition, the same address, the same static variables, inline for all or none, the same string literals
9. **[9.3/7]** - friend on members is possible after completion of class in which they are located. Example:
   ```
   class A {
   int foo();
   class B {
   friend A::foo();
   };
   };
   ```
   is not OK, whilst
   ```
   class C { int fword(); };
   class D {
   friend int C::fword();
   ```

```
};
```

is OK.

10. **[9.3/8]** - member of local classes must be defined inside (if defined at all)
11. **[9.3/9]** - only declarations, not definitions through typedef
12. **[9.3.1/1]** - it is not needed about the real type of object on which we perform member-call: `A *b; C *d; (A*)((void*)d)->f();` is undefined (supposing that A and C are not relatives, f() is method of A).
13. **[9.3.2/1-2]** `T x::id(params)` ⇒ `T id(const volatile X *this, params)`
14. **[9.3.2/3]** - constructor and destructor do not have cv-qualification
15. **[9.4/3]** - definition of static member shall not use directly names of non-static classes (only to form pointer to member)
16. **[9.4.1/2]** - static member function shall not be virtual. declaration of static member function shall not be cv-qualified
17. **[9.4.2/4]** - if static member is *const integral/enum* →
    ```
    class A { static const in a = 5; // this is decl };
    static const A::i; // !  this is definition
    ```
18. **[9.4.2/6]**
    (a) local class shall not have static data members
    (b) static data member of class in Namespace scope have extern
19. **[9.4.2/5]** - unnamed classes shall not contain static data members
20. **[9.4.2/8] -** static data member shall not be virtual
21. **[3.4.1/9]**; **[11.4.5]** friend functions defined in the class body have similar mechanism for searching as member functions of given class. Otherwise normal lookup, as for namespace functions.
22. **[8.3.6/3]** - default arguments only in
    (a) param_decl_clause of fun_decl (non-nested)
    (b) template_parameter
23. **[8.3.6/4]** - default arguments do not have "holes" →completion "backwards". They can't be redefined (even with the same value). Different scopes have different sets of default arguments.
24. So far, we know about two cases when declaration is "possibly not visible", those being: friend (**[3.4.1/3]**) and local extern declaration (**[3.3/4]**). It is necessary to distinguish these two, because Koenig search can see only friends (**[3.4.2/1]**) (double invisibility).
25. **[3.4.1/3]**
    (a) parameters are in scope function try block
    (b) alias declaration as in "f", "while", ...
26. If we find
    (a) goto label, we add it (with empty location) into table of labels (if it doesn't exist already).
    (b) label: we add location (and the label itself, if it doesn't exist already). If it already exists and it already has location, then it is a bug. It is also a bug if, after finishing parsing of function, there is a label without location.
27. **[3.2/3]** An inline f shall be defined in every translation unit in which it is used

28. **[3.2/4]** Definition of class is required if it's used in a way that needs class to be complete, which are the following cases:

    (a) definition of object
    (b) lval → rval applied to lvalue of T
    (c) conversion to T
    (d) conversion to T*, T&, all implicit conversions, dynamic cast, static cast (exception from NULL, void *)
    (e) member access to expression of type T
    (f) typeid, sizeof - operand is of type T
    (g) definition or call of f returning type T
    (h) lvalue of type T is assigned to

29. **[3.3.1]** points of declaration

    (a) **[/1]** immediately after declarator (before initializer)
    (b) **[/2]** hiding from point of declaration of the same name // Rudo
    (c) **[/3]** enumerator after enumerator_definition
    (d) **[/4]** class member can be looked up in the scope of it's class after point of declaration
    (e) **[/5]** point of declaration of class in elaborated_type_specifier (declaration using elaborated_type_specifier) Example:

```
int foo() {
struct XX {
struct D;
friend int g(struct B*);
int hh(struct D*);
friend struct B;
}
struct B;
struct C;
extern int g(struct B*);
}
struct C;
struct B;
int g(struct B);
```

    Notes: interacts with **[3.3.1],[3.3.1/5],[3.4.1/6]**

    This example shows that it is impossible to have the body of *g(...)*, because *struct B* in parameter is local class in foo, while *g* is present on namespace scope, from where it is impossible to "see" into *foo*.

    (f) **[/5]** Definition versus declaration. "namespace that contains declaration" - depending on interpretation, it is different to what *struct B* in the example below, point. We decided "declaration" (case number 1). Rationale: it would be madness to implement case numer 2.

```
namespace Q {
struct A {
int foo(struct B*);
}
struct B; // case # 1
}
int Q::A::foo(struct B*)();
struct B;
```

(g) Another contribution to "definition versus declaration" is in **[3.3.1/5]** and **[3.4.1/6]**. Declaration is considered as definition, according to these paragraphs (and only according to these paragraphs). Therefore `int A::foo(T);` is being searched in the same way as `int A::foo(T){}`

30. **[3.4.5/6]** If nested_name_specifier contains class template_id, ist is necessary for bison to switch us at the end of template arguments. (template arguments are searched in COE). It is not clear if it would be necessary to switch us also on the beginning (problem with guarding stack of contexts). // see lookup_class_member_acc (2)

31. **[3.5]** Plan for linkage:

    (a) check if internal_linkage
    (b) else check if it is not external linkage
    (c) otherwise no linkage
    (d) It is not clear if we can afford to set no linkage initially

32. **[3.6.1/3]** Main function

    (a) can't be called from the program itself (by programmer)
    (b) can't be declared neither with *inline* nor *static*
    (c) **[3.6.2/2]** describes argc, argv
    (d) linkage is implementation defined

33. **[3.6.2/4,5]** - termination. We probably have to know our own *exit()*, *return* jumps out and calls *exit(return_val)*

34. **[3.7.1/2,3]** - when handling both *static* and *automatic* objects, special care must be taken with respect to optimalizations (elimination according to **[12.8]**).

35. **[3.7.3.1/1]** (**allocation function - returns void *, has one parameter of type *size_t***) - allocation function can't be static, can't be defined in namespace different from global namespace, the first parameter can't have associated default argument, can be function template with two or more parameters.
    [3.3.3.1/2] `if (requested_size == 0) return non-NULL;`

36. **[3.7.3/2]** Shoudln't some kind of invisible declaration be added (to the beginning of translation unit), declaring *std::bad_alloc* to be able to report that it is being redefined by user?
    [3.7.3/3] When unsuccessful we report *std::bad_alloc* (or successor). If disabled, return only NULL (if it is disabled completely - *throw()*, if we can't *std::bad_alloc* (e.g. throw(int, double)) we nevertheless throw it. Contrary to it, we can't throw that *int* and *double*.

37. Every *delete, delete[]* returns void, has one parameter of type *void* **[3.7.3.2]**. Then when `delete p / delete[] p` is encountered, `operator delete(p) / operator delete[](p)` gets called respectively.

    (a) delete can have also parameter of type *std::size_t* which then gets the size of what has to be freed.
    (b) templated delete is never called by *delete p*, even if the second parameter would be *std::size_t*
    (c) templated delete must have 2 or more parameters (first one has to be void *, void has to be returned)
    (d) both *delete NULL* and *delete[] NULL* are OK
    (e) delete can be in in class or in global scope, otherwise is ill formed.

38. **[3.8]** Lifetime.
    Typical life of object:
    1) obtain storage
    2) call constructor
    3) lifetime
    4) call of destructors
    5) release storage
    Our implementation: reuse - constructor called on already allocated memory. release - deallocation without calling destructor **[3.8/5]**. Limited use of pointers before 2 and after 4. In 1 you can touch static data/member function. The similar applies to lvalues pointing to original data. Others have undefined behavior.
    Reuse - **[3.8/7]** - we decied that it will solve itself.

39. void **[3.9.1/9]**
    - empty set of values
    - incomplete f can't be completed
    - ! expression of type void can be used only as *expr_statement*, operand of ",", **[2.3]** of "?:", *typeid*, *return.*
    [**3.9.2/4]** (void *) the same representation and adjustment requirements as char *
    [3.9.3/3] const class - not involved with *static,* mutable and reference members. volatile - not involved with static reference members
    [3.10/13] const volatile - similar
40. **[3.10/5]** lvalues and rvalues
    (a) references are lvalues
    (b) functions returning reference yeild lvalue
    (c) functions having reference as their argument expect lvalue
    (d) **[3.10/6]** result from non-reference cast is r-value
    (e) **[3.10/9]** rvalue
        i. class - can have cv-qualification
        ii. non-class - always unqualified
    (f) rvalue always has completed type or void
41. Conversions **[4] [4.0/3]** Expression *e* can be implicitly converted to a type T ⇔ *T t* = *e* is well formed. The result is lvalue if T is reference type <u>and rvalue otherwise</u>.
    Conversions
    (a) **[4.1/1]** lval to rval
        i. only non-function, non-array
        ii. program which needs conversion of incomplete type is ill formed
        iii. non-class type "looses" cv-qualification
        iv. inside *sizeof* referenced value is not accessed
    (b) **[4.2]** array to pointer
        i. *type[n]*, *type[] - type*,* result points to the first element of the array
        ii. string literals - *char**, *wchar_t** - this is considered as sequence of two conversions, is not subsequence of lval-rval conversion (*"abc"* → *const char ** → *char **
    (c) **[4.3]** function to pointer
    (d) **[4.4]** qualification conversion - it is allowed to add to r-value pointer (member pointer respectively) cv-qualifications.
        In case of multilevel pointers, it is allowed to add cv-qualifier *a* to n-th element assuming that all "preceeding" elements will be *const.* Similarly for multilevel member pointer
        `int *** → int* volatile* const*` is OK
        `int *** → int *volatile**` is not OK
    (e) **[4.5]** Integral promotion
        i. signed/unsigned, char/short → int (or unsigned int, if int is not enough)
        ii. wchar_t, enumeration type → the first of int, unsigned int, long, unsigned long, to which the source fits
        iii. bitfield → int, unsigned int.
        iv. bool → int (true = 1, false = 0)
    (f) **[4.6]** floating promotion float → double
    (g) **[4.7]** integral conversions - **[4.9]** must read also jz (backend)!
    (h) **[4.10]** null-pointer constant either evaluates to 0 or can be converted to pointer type - Null pointer value, which is distinguishible from all other values of pointers to object
        Two null-pointers of the same type should compare equal
        NULL → cv qualified pointer is only single conversion
        For object types - *type** → *void ** - pointer is not modified. *cv D** → *cv B**, where B is base class. Program which requires this conversion to inaccessible or ambiguous *B* is ill-formed. Result is pointer to base class B. NULL pointer is converted to the NULL of type B.

(i) **[4.11]** ptr to member.
D →member ptr
D → cv member ptr
T B::* → T D::* (B ancestor of D)

(j) **[4.12]**

bool →arithmetic	0 = F, 1 = T
arithmetic → bool	0 = F, other = T
pointer → bool	NULL = F, other = T
member ptr → bool	NULL = F, other = T
enum → bool	0 = F, other = T

42. User defined conversions.
[12.3/4] There is at most one user defined conversion in implicit conversion sequence
[12.3/5] Sentences about "hiding" in this paragraph are being solved "atuomatically" by proper representation of structures we already have.

43. **[12.3.1/2]** explicit - changes behavior of conversion constructors → they are not seen when constructing implicit conversion (must be explicitely written)

44. **[12.3.1/1]** conversion constructors have exactly one parameter
[12.3.1/3] conversion constructors involve also copy-constructor (conversion to parent)

45. **[12.3.2/1]** it is not allowed to declare class, enum or typedef inside operator. It is not allowed to convert to the same type, type of predecessor, void, function type and array (for the last two, see **[12.3.2/4])**

46. Overload **[13]**.

(a) **[13.1/1]** - it is not allowed to have two function declarations (also using-declarations are forbidden (lists from lookup are not affected by this)) on the same scope, having the same name, which can't be overloaded

(b) **[13.1/2]** functions can't be overload if

i. they differ only in return type

ii. if exists static method with the same number and types of arguments
```
class A;
static T f();
Q f(); // error
static R h();
S h(int); // ok
static T g (A* thys);
Q g(); // ok
ZKONTROLOVAT, NEMUZU TO MOC PRECIST
```

(c) typedef-types are semantically (not syntactically) equivalent to their "expansions" (enum is not int).

(d) array is transformed to pointer: *[], [10]* → * (is not true for higher dimensions - *[][10] → (*)[10]*)

(e) function is transformed to pointer to function

(f) cv-qualifications - on outtermost level of parameter_type_spec the cv-qualification is ignored, i.e.
`int f(const int*)` $\not\equiv$ `int f(int *)`
`int f(int * const)` $\equiv$ `int f(int *)`

(g) differences in default arguments are ignored

47. **[11.4/9]** - friend in local class is being looked for only in innermost enclosing non-class scope

B.2 Rationales for Implementation Decisions

We don't mind invisible declarations and postponing of processing of function bodies, because (a) friend declarations are processed immediately (as they are not in function) (b) invisibles from local extern are not needed at all (they are not seen in *any* kind of lookup).

- the only reason for originating from local extern is to specify their namespace (and therefore also qualified name and *eo ipso* the name mangling)

  - With respect to the fact that invisibility originating from local extern is for any lookup, is it has no use for declaration tables (keeping on mind that there might be some troubles such as "different default arguments in different scopes", etc).
  - Because of these reasons is local extern put only into tables of local lookup.

- When sequentially elaborating declaration on function, the whole elaborated declaration is inserted into table. Rationale: simplification of overload resolutiom - we hold function with different default arguments "together". Simplification of finding "represenatative" declaration in the sense of "we have gathered together everything we know".
- translation unit is in fact compound statement (because of constructors of global variables) → declaration statement

Appendix C

Lookup Algorithms

C.1 Qualified Name Lookup

There

- **[3.4.3/1]**

A	::	B
object,function enumerator_names ignored		name nominating member of class or namespace

if name which was found is not class or namespace name, then the program is ill-formed.

- **[3.4.3/3]**
  if (decl_id) is qualif_id, then in case of α*decl_id* β - for α lookup in defining namespace scope is performed, for β lookup in scope of member's class (namespace) is performed (when performing reduction of decl_id it is necessary to switch context)

- **[3.4.3/4]** unary :: lookup in global scope of translation unit through (a) declaration (b) using-directive

- **[3.4.3/5] Pseudo destructors (PDN)**
  type names are looked up (as types) in scope designated by nested-name-specifier, e.g:
  `C::I::~I()`

- **[3.4.3.1]** Class
  If nested-name-specifier of qual_id nominates class, name after nested-name-specifier is looked up in scope of the class resulting in either member(s) of member(s) of base-class.

  The case *A::A* - lookup leads to constructor; it can be used only in declarator-id of construction def:**???**(outside of class-def).

  Exceptions are:

  - pseudodestructors
  - conversion-type-id of oper-fun-id looked up also in contest in which entire expression occurs. Both lookups shall refer to the same type
  - template arguments of templ_id are looked up in context in which entire expression occurs.

C.1.1 Qualified Lookup Algorithm

The function *qlookup* looks up the name in context for the purposes of qualified lookup.

function *qlookup*(*name*,*content*,*in_decl*,*before_qdot*) : **set of** *declarations***;**
begin
assert(*context is namespace*) // **[3.3.2.2/6]**
case (⟨*in_decl*, *before_qdot*⟩) **of:**
⟨true, false⟩: **return** *qlhelper*(*name*, *context*);

```
    ⟨true,true⟩:
     s = qlhelperCN(name,context)
     // this part is controversial according to interpretation of [3.4.3.2/6]
     if s ≠ 0 then
       for each c in usings_of(context) do
         s ∪ = qlookup(name,c,true,true)
       done
     fi
     // end of controversial part
     return s;
    ⟨false, false⟩:
     s = qlhelper(name,context);
     if s ≠ 0 then
       for each c in usings_of(context) do
         s ∪ = qlookup(name,c,false,false)
       done
     fi
     return s;
    ⟨false, true⟩:
     s = qlhelperCN(name,context)
     if s ≠ 0 then
       for each c in usings_of(context) do
         s ∪ = qlookup(name,c,true,true)
       done
     fi
     return s;
    esac
  end.
```

function *qlhelper*(*name*,*context*):**set of declaration** ... returns declaration of name declared directly inside the context.
function *qlhelperCN*(*name*,*context*):**set of declaration** ... is the same as the previous one, but ignores enumarators, objects and functions **[3.4.3/1]**

C.2 Koenig

- we see even invisible friends
- typedef and usings do not contribute

C.2.1 Finding of associated classes and namespaces for type T

function *koenig_associates*(*T*:*TYPE*): **pair of (set of classes) and (set of namespaces)**
case (T) **of:**
fundamental_type: return $\langle \emptyset, \emptyset \rangle$;
class_type: $classes_0 \leftarrow \{T\}$
$classes \leftarrow fixedpoint_x(X = X \cup \{P | P \text{ is ancestor of } x \in X\} \cup classes_0)$
$classes \cup = \{class\ ennclosing\ T\}$
$namespaces \leftarrow \{N | N = namespace_of(\exists x \in classes)\}$
// i.e. for all classes add its namespace
return $\langle classes, namespaces \rangle$
enumeration:
$namespaces \leftarrow \{namespace_of(T)\}$
$classes \leftarrow \{classes\ in\ which\ T\ is\ defined\}$
return $\langle classes, namespaces \rangle$
pointer,array: let *U* **be** *type which is pointed to/element of array*

return *koenig_associates*(U);
function:
return $\langle \cup\{(koenig_associates(x))_{classes} | x \in (argtypes(T) \cup \{rettype(T)\})\},$
$\cup\{(koenig_associates(x))_{namespaces} | x \in (argtypes(T) \cup \{rettype(T)\})\}\rangle$
member pointer: *member pointer T points to U in class X*
$\langle a,b\rangle \leftarrow koenig_associates(U)$
$\langle c,d\rangle \leftarrow koenig_associates(X)$
return $\langle a \cup c, b \cup d\rangle$
template-id:
$cl \leftarrow \{class\ in\ which\ is\ defined\ as\ template\}$
$n \leftarrow \{namespace\ in\ which\ T\ is\ defined\}$
$cl\cup = \cup\{(koenig_associates(x))_{classes} | x \in (arguments \backslash templ_templ_args \backslash non_type_args)\}$
$n\cup = \cup\{(koenig_associates(x))_{namespaces} | x \in (arguments \backslash templ_templ_args \backslash non_type_args)\}$
$cl\cup = \{class\ in\ which\ X\ is\ defined | X \in templ_templ_args\}$
$cl\cup = \{namespace in which X is defined | X \in templ_templ_args\}$
return $\langle cl, n\rangle$
if *T is non-type* **then return** $\langle\emptyset,\emptyset\rangle$; **fi**
[3.4.2/2, second paragraph (in addition...)]

In the case of function argument pointing to set (or & of set) either overloaded functions (lookup without koening, we don't have arguments, only **???** = **Nk)** or templates of functions.

```
f(blah):
Nk.f = {f_1, ... , f_k}
Now call Koenig(f) → Nk bla = {bla_1, ..., bla_k}
return (Ck, Nk bla) ∪  (0, {∃ x ∈ Nkbla & x ∈ N}) // map + union
```

Koenig_associates called for types of arguments. If the type of argument is not known to us (which can legally happen only for argument of type "foo" or "&foo", where foo is overloaded (> 1) function, we proceed as follows:

For every possible found type (i.e. signature of function "foo") we find *Koening_associates,* and unite them. But if there is, by any chance, between "signatures of foo" signature of template S, we have (el,n) by definition as follows:

```
n = {ns, in which this foo is already declared}
cl ∪ = {(Koenig_as(x))_cl}; x ∈ types of parameters(S) \
types of parameters depending on parameters of S} + return type is independent
```

Example:

```
template< class X > X foo (int, X*, ::std::list<int>*, ::std::list<x>*);
template<class X> ::std::string* bar(X*);
```

The underlined ones are independent and therefore are considered in previous paragraph.

This is definitive end of Koenig_associates.

```
[3.4.2/2a]
function Koenig(X:ident, unql_found:decl_list):decl_list
if((∃ x ∈ unql_found)(x is member function))
        return unql_found;
else {
        (ass_cl, ass_ns) ← Koenig_associtès(unql_found);
```

"lookup in ass_cl, ass_ns" similarly as one level Qname lookup, **but**:

- ignore using_directive
- ignore invisibility of friends

C.3 Lookup_class member access I

[3.4.5] Postfix expression (*a*)→ id_expression (*b*)
Handling of templates **[3.4.5/1]**

```
x = lookup(b in class A);
if (x==0) → g = lookup(b in context of entire expr)
        if (g ∉ {class_templ,func_templ}) ⇒ ill-formed
        return g;
if (x is template) → g = lookup(b in context of entire expr)
        if (y==0) ⇒ return x;
        if (y isn't class_templ) return x;
        if (x==y) ⇒ return x;
        else ill-formed;
```

id_expression is unqualified_id

```
if(a is scalar) ⇒
          (x = lookup(b in ctxt of entire expr))
else
          x = lookup(b in class A)
return X
```

Destructors:

```
if (a is class_type) {
 if (b is ~typename) {
   B = typename;
   if (B isn't classtype) ⇒ ill-formed
   x = lookup(B in A)
   g = lookup(B in COE)
   if (x!=y) ⇒ ill-formed
 }
if (a is  scalar type) ⇒ x = lookup (B in COE)
return X;
}
```

Qualified-id, which looks like "class-or-ns-name (Γ) :: nested-name-spec **???** unq-id" (ψ). Γ is looked up as follows:

```
x = lookup (Γ in A);
assert(|x| ≤ 1 & (∀a ∈ x)(class-name-p(a))
y = lookup(Γ in COE);
assert((∀a ∈ y)(class-name-p(a))
if (x == y) return lookup (ψ in A);
if (x == 0) return lookup (ψ in COE);
if (y == 0) return lookup (ψ in A);
ill-formed;
```

Qualified id "::nested-name-spec unqualif-id" (ψ)

```
return lookup(ψ in global_scope)
```

[3.4.5/6] arguments of template are evaluated in COE → this is also caveat, because it is neccessary to rewrite hinetr.

Conversion function id "operator type" (ψ)

```
assert(lookup(ψ in COE) ≡ lookup (ψ in A))
return lookup (ψ in A)
```

class template id → see caveats

C.4 Lookup - elaborated_type_specifiers, using_directive

[3.4.4/2] (ETS_name is simple identifier)

- if elaborated_type_spec is of form “classkey ident”
  lookup according to unqualified name lookup
  - ignore non-type names
  - name is typedef-name ⇒ ill-formed
  - ETS refers to enum && this enum is not previously-declared ⇒ ill-formed
  - ETS refers to class-name ⇒ introduce name (**[3.3.1],** and also point (29) in Caveats)
- “class_key ident”; if refers to class-name ⇒ introduce

(ETS_name is qualified)

- lookup according to qualified lookup
  - ignore nontype names
  - typedef ⇒ ill_formed
  - class_name, enum_name ⇒ OK
  - otherwise ill-formed

no introduction. **[7.1.3.5]** class u**?**; ⇒ ill-formed
[3.4.6] lookup for name in

- using_directive
- namespace_alias_definition

lookups only for namespace names

C.5 Lookup⇔ Hinter Interaction

Questions for hinter (questions of different type from “what is this identifier”):

- ? - bit fields, integral type
- ? - constructors
- ? -

constructor-decl → function_specifier_seq cname.

- cname → (cname)
- cname → name_of_this_class

We have hinter for names. Oracle with hints.

Appendix D

Implementation definitions

All decisions below are implementation defined.

D.1 Reference(accessing expression through &) does not constitute an access to volatile variable.

Rationale: Difficulties in determining of volatile access. References: - info '(gcc)Volatiles' - ss_access_of.has_se TODO: Warning generation.

D.2 Dereference of pointer to volatile doesnot constitue an access to volatile object.

D.3 Conversion of volatile lvalue(vol_get) to an rvalue constitutes an volatile access, hence a side effect.