Implementation of the Cmajor Compiler

Seppo Laakko

June 20, 2016

Contents

Chapter 1

Introduction

This document describes the implementation of the Cmajor compiler front-end. We also inspect some excerpts of language theory and parsing theory as we go on to make the description of implementation hopefully more understandable.

1.1 Cmajor Programming Language and Cmajor Compilers

Cmajor is a hybrid programming language that combines C# like syntax with C++ like semantics. The original Cmajor compiler is written in C++. Now there is also a Cmajor compiler written in Cmajor that was created by manually converting the C++ version to Cmajor. However it still lacks some features that are present in the C++ version, so the the principal version as of this writing remains to be the C++ version.

1.2 Phases of Compilation

In classical compiler text books the compilation consists in principle of the following phases:

1. In the lexical analysis phase a stream of characters of source code of a program is broken into lexical units called *lexemes* and an integer or enumerated value called a *token* is assigned to each lexeme.
2. In the syntax analysis phase the grammatical structure of tokens are analyzed, and *abstract syntax trees* are generated.
3. In the semantic analysis phase the syntax trees are traversed and the program is type-checked and verified that it consists of semantically meaningful elements.
4. In the intermediate code generation phase intermediate code for program elements are generated.
5. In the machine-independent code optimization phase intermediate code is processed and optimized using various passes.
6. In the code generation phase machine code is generated.
7. In the machine-dependent code optimization phase the machine code is optimized further and target machine code is generated.

The compiler collects information[1] about identifiers encountered in the program into a *symbol table* and consults the symbol table when information about an identifier is needed.

Example 1.2.1. Consider the following source code fragment:

```
1 x = 10 * x + (cast<int>(c) - cast<int>('0'));
```

We are now going to have a taste of what the input and output of each phase of the compilation looks like.

1. Lexical analysis. The lexical analyzer might produce the following lexemes for the code fragment above:

   `x`, `=`, `10`, `*`, `x`, `+`, `(`, `cast`, `<`, `int`, `>`, `(`, `c`, `)`, `-`, `cast`, `<`, `int`, `>`, `(`, `'0'`, `)`, `)` and `;`.

   If we represent punctuation and other symbolic lexemes with token values equal to themselves and other lexemes with upper case identifiers, the lexical analyzer may assign the following tokens to the lexemes that do not represent themselves:

   - `x` : **ID** (identifier)
   - `10` : **INTLIT** (integer literal)
   - `cast` : **CAST** (reserved word)
   - `int` : **INT** (reserved word)
   - `c` : **ID** (identifier)
   - `'0'` : **CHARLIT** (character literal)

2. Syntactic analysis. The syntax analyzer or *parser* receives the following token stream from the lexical analyzer or *lexer*:

   ID, `=`, **INTLIT**, `*`, **ID**, `+`, `(`, **CAST**, `<`, **INT**, `>`, `(`, **ID**, `)`, `-`, **CAST**, `<`, **INT**, `>`, `(`, **CHARLIT**, `)`, `)` and `;`.

   The result of phase 2 is an abstract syntax tree or *AST* that reveals the syntactic structure of the source code. Thus the parser may produce the following abstract syntax tree for the code fragment:

```
AssignmentStatementNode
    IdentifierNode(x)
    AddNode
        MulNode
            SByteLiteralNode(10)
            IdentifierNode(x)
        SubNode
            CastNode
                IntNode
                IdentifierNode(c)
            CastNode
                IntNode
                CharLiteralNode('0')
```

[1] type for example

3. Semantic analysis. The abstract syntax trees generated in phase 2 are traversed and the program is type-checked. Assuming that identifier `x` has been declared earlier to be a variable of type **int** and identifier `c` to be a variable of type **char**, the type-checker finds this information in the symbol table, when it walks the syntax tree.

   When encountering the `MulNode` the type-checker checks whether it is legal to multiply an **sbyte** literal 10 by a variable x of type **int**. This is the case so it records that the result of this multiplication produces a value of type **int**.

   When encountering the first `CastNode` it checks if it is legal to convert a variable c of type **char** to type **int**. Similarly for the second `CastNode`, the conversion of the character literal '0' to type **int** is checked. They are both legal so the `SubNode` produces a value of type **int**.

   When encountering the `AddNode` two **int** values are added and the result is of type **int**.

   Finally when encountering the `AssignmentStatementNode` the type-checker checks whether it is legal to assign a value of **int** to a variable `x` of type **int**. This is the case so the type-checking succeeds.

4. Intermediate code generation. The following intermediate code[2] may be produced from the abstract syntax tree and from information stored in the symbol table:

```
%1 = sext i8 10 to i32
%2 = load i32, i32* %x
%3 = mul i32 %1, %2
%4 = load i8, i8* %c
%5 = zext i8 %4 to i32
%6 = zext i8 48 to i32
%7 = sub i32 %5, %6
%8 = add i32 %3, %7
store i32 %8, i32* %x
```

   Quick introduction to intermediate instructions:

   - `%1`, `%2`, etc. represent intermediate results of computation. They may be regarded as registers. There are inifinite number of them.
   - **i8**, **i16** and **i32** are 8-bit, 16-bit and 32-bit integer types.
   - **sext** instruction *sign extends* its operand to a target type.
   - **load** instruction loads a value of a variable.
   - **mul** instruction multiplies two values.
   - **zext** instruction *zero extends* its operand to a target type.
   - **sub** instruction subtracts a value from another.
   - **add** instruction adds two values.
   - **store** instruction stores a value to a variable.

[2] this is LLVM intermediate code [4]

5. Code optimization. The following optimized intermediate code may be generated from the intermediate code produced in phase 4:

```
%1 = load i32, i32* %x
%2 = mul i32 %1, 10
%3 = load i8, i8* %c
%4 = zext i8 %3 to i32
%5 = add i32 %2, -48
%6 = add i32 %5, %4
store i32 %6, i32* %x
```

6. Machine code generation. The following fragment of assembly code may be generated:

```
movl 8(%rsp), %eax
leal (%rax,%rax,4), %eax
movzbl 7(%rsp), %ecx
leal -48(%rcx,%rax,2), %eax
movl %eax, 8(%rsp)
```

1.3 Front-end and Back-end of a Compiler

The lexical, syntactic and semantic analysis phases and intermediate code generation phase form a *front-end* of a compiler. The optimization and target machine code generation phases form a *back-end* of a compiler.

By combining N programming language specific front-ends with M target machine architecture specific back-ends it is possible to create N times M compilers by writing only N plus M programs.

Intermediate code is the glue between the front and back ends of a compiler.

1.4 Representations of Cmajor Programs

The Cmajor compiler front-end has many intermediate representations for Cmajor programs:

- The first one is the *abstract syntax tree representation* that the *parser* component produces. It reflects faithfully the syntactic structure of the Cmajor source code. In this representation identifiers do not yet refer to any symbol, they are just identifiers.
- The second one is the *bound tree* representation that is a high-level intermediate representation. The top-level of bound node hierarchy there are bound compile units. The next level is formed by bound classes and bound functions. In the lowest level there are bound node types for different kinds of statements and expressions. In this representation all identifiers have been bound to refer to a specific symbol: a variable, parameter, constant, enumeration constant, type or function symbol, for example. The bound tree is produced from the abstract syntax tree by the *binder* component.
- Finally there are two low-level intermediate representations: the LLVM and C representations that are produced from the bound tree representation by the *emitter* component.

1.5 The Structure of This Document

The structure of the rest of this document is as follows:

- Chapters 2 and 3 are devoted to the lexical and syntactic analysis phases of compilation. We go through some theory and then see how they are implemented in the parser component of the Cmajor compiler.
- Chapter 4 describes the hierarchial symbol table component and presents algorithms for name lookup and construction of the symbol table.
- In chapters 5, 6 and 7 we inspect three utility components used by other components of the compiler: the type repository, the static evaluator and the type resolver.
- The following eight chapters descibe different aspects of the binder component:
  - We begin by binding types and values in chapter 8.
  - Chapters 9, 10 and 11 describe function overload resolution:
    - Chapter 9 presents function repositories.
    - In chapter 10 we inspect the algorithms used in overload resolution.
    - Chapter 11 discusses concepts and their role in the overload resolution.
  - Chapter 12 describes the operation of the expression binder and in chapter 13 we inspect various statement binders.
  - In chapter 14 we concentrate on how templates are compiled.
  - Chapter 15 bundles the previous pieces together by describing the binding of whole compile units.
- Chapter 16 is devoted to the emitter component of the compiler

Chapter 2

Lexical Analysis

The first phase of compilation is to break the character stream into tokens that are passed along to the parser. Here a token is defined to be a name and an attribute value. For example, **INTLIT** with a value 10.

Typically these tokens are described as *patterns* that define the form that the lexemes of a token may take. Here a lexeme is the actual sequence of characters in an input stream that match that pattern. One way to describe those patterns is to use *regular expressions*.

2.1 A Bit of Language Theory

To describe regular expressions we take a small break and define a few fundamental concepts.

2.1.1 Alphabets

An *alphabet* is a finite, nonempty set of symbols. Conventionally, we use the symbol Σ for an alphabet ([2] pg. 28).

Typical alphabets are:

- $\Sigma = \{0, 1\}$, a binary alphabet.
- $\Sigma = \{a, \ldots z\}$, the alphabet of lowercase latin letters.
- The set of ASCII characters.
- The set of Unicode characters.

2.1.2 Strings

A *string* is a finite sequence of symbols chosen from some alphabet ([2] pg. 29). An *empty string* is the string of zero occurrences of symbols. It is denoted ϵ.

2.1.2.1 Powers of an Alphabet

If Σ is an alphabet, we define Σ^k to be the set of strings of length k, each of whose symbols is in Σ ([2] pg. 29).

Thus if $\Sigma = \{0, 1\}$, the binary alphabet:

- $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

The set of all strings over an alphabet is denoted Σ^*.

2.1.3 Languages

A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a *language* ([2] pg. 30). If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ.

Examples of languages:

- English: the collection of legal English words is a set of strings over the alphabet that consists of all the letters.
- The language of legal C programs: the alphabet is a subset of ASCII characters, and the language is a subset of all possible strings over that alphabet.
- The set of binary numbers whose value is prime:

$$\{10, 11, 101, 111, 1011, \ldots\}$$

- $\emptyset$, the empty language, is a language over any alphabet.
- Σ^* is a language over any alphabet.
- The language of all possible UTF-8 encoded strings of Unicode characters, denoted L_{UTF8}.
- The language of syntactically valid Cmajor programs, $L_{Cmajor} \subset L_{UTF8}$.

2.1.4 Regular Expressions

Regular expressions define languages.

Before describing the notation of regular expressions, we need to define three operations on languages that the operators of regular expressions represent:

1. The *union* of two languages L and M, denoted $L \cup M$, is the set of strings that are in either L or M, or both ([2] pg. 84). For example, if $L = \{01, 10\}$ and $M = \{10, 100\}$, $L \cup M = \{01, 10, 100\}$.
2. The *concatenation* of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M ([2] pg. 84). We denote concatenation of L and M LM. For example, if $L = \{01, 10\}$ and $M = \{10, 100\}$, $LM = \{0110, 01100, 1010, 10100\}$.
3. The *closure* of a language L, denoted L^*, is the infinite union $\cup_{i \geq 0} L^i$, where $L^0 = \{\epsilon\}$, the set containing the empty string, $L^1 = L$, and L^i, for $i > 1$, is $LL \cdots L$, the concatenation of i copies of L ([2] pg. 85). For example, if $L = \{01, 10\}$, $L^* = \{\epsilon, 01, 10, 0101, 0110, 1010, \ldots\}$. That is: L^0 gives $\{\epsilon\}$, the empty string, $L^1 = L$ gives $\{01, 10\}$; $L^2 = LL$ gives $\{0101, 0110, 1001, 1010\}$ and so on.

Now regular expressions can be defined recursively as follows:

BASIS: There are three parts:

1. The constants ϵ and $\emptyset$ are regular expressions that denote languages $\{\epsilon\}$ and $\emptyset$ respectively. That is, $L(\epsilon) = \{\epsilon\}$ and $L(\emptyset) = \emptyset$.
2. If a is any symbol, then $\mathbf{a}$ is a regular expression [1]. This regular expression denotes the language $\{a\}$. That is, $L(\mathbf{a}) = \{a\}$.
3. A variable L represents any language.

INDUCTION: There are four parts:

1. If E and F are regular expressions, then $E|F$ is a regular expression that denotes a union of $L(E)$ and $L(F)$. That is, $L(E|F) = L(E) \cup L(F)$.
2. If E and F are regular expressions, then EF is a regular expression that denotes the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.
3. If E is a regular expression, then E^* is a regular expression that denotes the closure of $L(E)$. That is, $L(E^*) = (L(E))^*$.
4. If E is a regular expression, then (E), a parenthesized regular expression, is also a regular expression, that denotes the same language as E. That is, $L((E)) = L(E)$.

Example 2.1.1. Let us use the formal theory to build a regular expression for sequence of one or more decimal digits. First we use the basis rule 2 to build regular expressions for decimal digits:

$$\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}$$

Now we have languages

$$L(\mathbf{0}) = \{0\}, \ldots, L(\mathbf{9}) = \{9\}$$

Next we use induction step 1 to build a regular expression for any decimal digit, denoted by D:

$$D = \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}|\mathbf{6}|\mathbf{7}|\mathbf{8}|\mathbf{9}$$

Now we have a language for a single decimal digit:

$$L(D) = \{0, 1, \ldots, 9\}$$

Next we use induction step 3 to build a regular expression of any number, including zero, decimal digits:

$$E = D^*$$

Now we have a language for any number of decimal digits:

$$L(E) = \{\epsilon, 0, 1, \ldots, 9, 00, 01, \ldots, 09, \ldots\}$$

Finally we exclude the empty string by concatenating one decimal digit with any number of decimal digits:

$$F = DD^*$$

The language for nonempty sequence of decimal digits is thus

$$L(F) = \{0, 1, \ldots, 9, 00, 01, \ldots, 09, \ldots\}$$

[1] Here we denote regular expressions using **bold typeface** and symbols using *italics*.

2.2 Tools for Lexical Analysis

Regular expressions can be used to describe patterns that form tokens. But using regular expressions, one can describe only relatively simple kind of languages, namely *regular languages*.

Strings that belong to a particular regular language can be recognized by constructing a *finite automaton*. A finite automaton is a kind of *state machine*, it has states and transitions between the states, but it has limited "memory". It cannot for example recognize the language of arbitrary long strings of balanced parentheses.

Many fundamental programming language constructs such as identifiers and literals are regular, but to recognize potentially infinitely deep block structures, one needs to have a more powerful kind of language recognizer, a finite automaton with a stack, or a *pushdown automaton*.

A pushdown automaton can recognize a language that is *context-free*. The languages for syntactic structures in many programming languages are mostly context-free, but for some constructs one may need to provide lexical information to guide the parser.

Finite automata can be constructed by hand, but there are also tools that take regular expression patterns as input and construct a lexical analyzer that recognize those patterns. Such a tool is called a *lexical-analyzer generator*. Most famous is the Unix tool `lex` and its GNU version `flex`.

2.3 Lexical Analysis in Cmajor

The Cmajor compiler includes a tool called Cmajor Parser Generator, `cmpg`, that combines the role of a parser generator and a lexical-analyzer generator, or more truly, it is a parser generator that can be used without the need to have a separate lexical-analyzer generator.

2.3.1 Introduction to Cmajor Parser Generator

The following table summarises some `cmpg` expressions:

Expression	Matches	Example
empty	empty string	**empty**
space	any white space character	**space**
anychar	any single character	**anychar**
letter	any latin letter	**letter**
digit	any decimal digit	**digit**
hexdigit	any hexadecimal digit	**hexdigit**
punctuation	any ASCII punctuation character	**punctuation**
'c'	character c	'a'
\c	character c literally	\(
"s"	string s	"0x"
[s]	any one of characters in s	[abc]
[^s]	any one character not in s	[^abc]
$r*$	zero or more strings matching r	a*
$r+$	one or more strings matching r	a+
$r?$	zero or one r	a?
$r_1 r_2$	an r_1 followed by an r_2	ab
$r_1 \mid r_2$	an r_1 or an r_2	a|b
$r_1 - r_2$	r_1 but not r_2	**anychar** − "*/"

To use `cmpg`, one prepares *.parser* files that contain `cmpg` grammar definitions, and a *.pp* file that lists the *.parser* files, and issues a command

```
cmpg file.pp
```

The `cmpg` reads and validates the grammar definitions in the *.parser* files and generates a C++ source and header files that contain C++ classes for each defined grammar. When the resulting C++ source files are compiled and linked with *Cm.Parsing* library, the result is a top-down backtracking parser.

2.3.2 Tokens in Cmajor

We are now going to take a look of some classes of tokens in Cmajor programming language, and how they are defined using `cmpg` expressions.

2.3.2.1 Skipping Whitespace and Comments

We are not interested in contents of comments or whitespace during parsing, so they are skipped. In a `cmpg` grammar, one can define a *skip* clause, to set a *skip rule* that is in effect during parsing. The parser alternates between parsing other tokens and skip tokens. In the main compile unit grammar the skip rule is set to spaces_and_comments rule:

```
1  grammar CompileUnitGrammar
2  {
3      // ...
4      skip spaces_and_comments;
5      // ...
6  }
```

The *spaces_and_comments* rule is defined here. Note that the end of the block comment, "*/", is not matched inside string or character literals.

```
1  spaces_and_comments
2      ::= (space | comment)+
3      ;
4
5  comment
6      ::= line_comment | block_comment
7      ;
8
9  line_comment
10     ::= "//" [^\r\n]* newline
11     ;
12
13 newline
14     ::= "\r\n" | "\n" | "\r"
15     ;
16
17 block_comment
18     ::= "/*" (StringLiteral | CharLiteral | (anychar - "*/"))* "*/"
19     ;
```

2.3.2.2 Identifiers and Keywords

When parsing an identifier, for example, we must disable the skip rule. Otherwise the parser would accept string "iden ti fier" as an identifier, because whitespace is skipped. For that, the `cmpg` language has a **token** expression. The **token** expression suppresses the skip rule when parsing the contents of the expression.

The difference expression, $r_1 - r_2$, matches r_1 but not r_2. In this case $id\_chars - Keyword$ in line 2 rejects keywords as identifiers.

The **keyword_list** expression in line 10 has two components. The first is a name of a rule that selects a token, in this case *id_chars*, and the second is a list of keyword strings that are matched against the selected token. If the selected token is found among the keyword strings, the **keyword_list** expression accepts the selected token, otherwise it rejects it.

```
1  Identifier
2      ::= token(id_chars - Keyword)
3      ;
4
5  id_chars
6      ::= token((letter | '_') (letter | digit | '_')*)
7      ;
8
9  Keyword
10     ::= keyword_list(id_chars,
11         ["abstract", "and", "as", "axiom", "base", "bool", ...,
12          "where", "while"])
13     ;
```

2.3.2.3 Literals

Literals in Cmajor, as in many other programming languages, can be parsed with regular expressions.

- Let us start one of the simplest, a Boolean literal:

```
1  BooleanLiteral
2      ::= keyword("true")
3      |   keyword("false")
4      ;
```

  The **keyword** expression matches the input to its parameter string, but it accepts the input only if the input does *not* continue with an identifier character: a letter, a digit or an underscore. If the *BooleanLiteral* rule were defined using plain strings, like this:

  ```
  BooleanLiteral ::= "true" | "false"
  ```

  input like "truely" or "falsely" would be accepted as a *BooleanLiteral* followed by "ly" suffix. This is not what we want, so we use the **keyword** expression.

- Floating point numbers have many forms. The *fractional_real* rule accepts inputs having a fractional part like "1.23", ".987", "1.23e3" and "3.". The *exponent_real* rule accepts decimal digits followed by exponent part like "1e-2".

```
FloatingLiteral
    ::= token((fractional_real | exponent_real)('f' | 'F')?)
    ;

fractional_real
    ::= token(digit_sequence? '.' digit_sequence exponent_part?)
    |   token(digit_sequence '.')
    ;

digit_sequence
    ::= token(digit+)
    ;

sign
    ::= '+' | '-'
    ;

exponent_real
    ::= token(digit_sequence exponent_part)
    ;

exponent_part
    ::= token([eE] sign? digit_sequence)
    ;
```

An optional 'f' or 'F' suffix denotes floating point literal that has type **float**. Without the suffix floating point literals have type **double**.

- An integer literal can have either hexadecimal or decimal form. The "0x" or "0X" prefix denotes hexadecimal integer literal.

```
IntegerLiteral
    ::= (hex_literal | digit_sequence) ('u' | 'U')?
    ;

hex_literal
    ::= token(("0x" | "0X") hex)
    ;

hex
    ::= token(hexdigit+)
    ;
```

In Cmajor the type of an integer literal is the first of the of the following types in which its value can be represented: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**.

The 'u' or 'U' suffix denotes an integer literal with an unsigned type. The type of it is the first of the following types in which its value can be represented: **byte**, **ushort**, **uint**, **ulong**.

- The character literal rule accepts regular characters like 'a' or 'X', simple escapes like '\n' and '\r', hexadecimal escapes like '\xef', and decimal escapes like '\d100'. Other escaped characters represent themselves.

```
CharLiteral
    ::= token('\'' ([^\\\r\n] | escape)'\'')
    ;

escape
    ::= token('\\' ([xX] hex | [dD] digit_sequence | [^dDxX]))
    ;
```

- String literals can have four forms.

  1. Regular strings like "abc", or strings containing escaped characters like "line\n". The type of regular string literal is **const char***.
  2. Wide strings like w"abc", or wide strings containing escapes. The type of wide string literal is **const wchar***.
  3. Unicode strings like u"abc", or Unicode strings containing escapes. The type of Unicode string literal is **const uchar***.
  4. Raw strings, that have @-prefix and have no escapes in them, like @"abc\". The contents of raw string is taken literally. The type of raw string literal is **const char***.

```
StringLiteral
    ::= string
    |   'w'string
    |   'u'string
    |   raw_string
    ;

string
    ::= token('"' (([^"\\\r\n]+) | escape)* '"')
    ;

raw_string
    ::= '@' token('"' [^"]* '"')
    ;
```

- The last literal is the simplest, it's the null literal:

```
NullLiteral
    ::= keyword("null")
    ;
```

Chapter 3

Syntax Analysis

We are now going to explore a class of languages that are suitable for defining the grammatical structure of a programming language, namely *context-free languages*. Context-free languages extend the notion of regular languages so that with a context-free language one can express also recursive structures like nesting blocks or balanced parentheses.

3.1 Example

Example 3.1.1. A *palindrome* is a string that reads the same forward or backward, such as `otto` or `madamimadam` ("Madam, I'm Adam", the first words that Adam said to Eve in the Garden of Eden.) We can define palindromes for the binary alphabet, $\Sigma = \{0, 1\}$, recursively as follows:

BASIS
ϵ, i.e. the empty string, 0, and 1 are palindromes.

INDUCTION
If P is a palindrome, so are $0P0$ and $1P1$. No string is a palindrome of 0's and 1's unless it follows from this basis and induction rule.

A context-free grammar is a formal notation for expressing such recursive definitions of languages ([2] pg. 170). A grammar consists of one or more variables that represent classes of strings, i.e. languages. In previous example we have only one variable, P, which represents the set of palindromes; that is the class of strings forming the language L_{pal}. There are rules that say how the strings in each class are constructed. The construction can use symbols of the alphabet, strings that are known to be in one of the classes, or both.

Grammar 3.1.1. The rules that define the palindromes, expressed in the context-free grammar notation, are:

$$P \to \epsilon \tag{3.1}$$
$$P \to 0 \tag{3.2}$$
$$P \to 1 \tag{3.3}$$
$$P \to 0P0 \tag{3.4}$$
$$P \to 1P1 \tag{3.5}$$

The first three rules form the basis. They tell us that a class of palindromes includes the strings ϵ, 0, and 1. None of the right sides of these rules contains a variable, which is why they form a basis for the definition.

The last two rules form the inductive part of the definition. For instance, rule 3.4 says that if we take any string ω from the class P, then $0\omega0$ is also in class P. Rule 3.5 likewise tells us that $1\omega1$ is also in class P.

3.2 Definition of Context-Free Grammars

There are four important components in a grammatical description of a language ([2] pg. 171):

1. There is a finite set of symbols that form the strings of the language being defined. This set was $\{0, 1\}$ in the palindrome example. We call this alphabet the *terminals*, or *terminal symbols*.

2. There is a finite set of *variables*, sometimes called *nonterminals*. Each variable represents a language; i.e. a set of strings. In the last example, there was only one variable, P, which we used to represent the class of palindromes over alphabet $\{0, 1\}$.

3. One of the variables represents the language being defined; it is called the *start symbol*. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol. In our example, P, the only variable, is the start symbol.

4. There is a finite set of *productions* or *rules* that represent the recursive definitiion of the language. Each production consists of:

   (a) A variable that is being (partially) defined by the production. This variable is often called the *head* of the production.

   (b) The production symbol $\rightarrow$.

   (c) A string of zero or more terminals and variables. This string, called the *body* of the production, represents one way to form strings in the of the variable of the head. In doing so, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

We follow a convention that if the start symbol is not explicitly specified, the head of the first production of the grammar is the start symbol.

3.2.1 Derivations Using a Grammar

To infer that a certain string is in the language of a grammar, we start with the start symbol of the grammar and expand it using one of its productions, i.e. by replacing the head of the production with its body. Then we further expand the resulting string by replacing one of its variables by the body of one of its productions, and so on, until we derive a string consisting entirely of terminals. The language of the is all strings of terminals that we can obtain this way. This use of grammar is called a *derivation*.

To see that string 0110 is in the language of binary palindromes L_{pal}, for example, we start from the start symbol P, and replace it with the body of the production 4 of grammar 3.1.1:

$P \Rightarrow 0P0$. We then replace the variable P between the 0's with the body of the production 5: $0P0 \Rightarrow 01P10$. Finally we replace the variable P in the obtained string with the body of the production 1: $01P10 \Rightarrow 01\epsilon10$. That way we have the derivation $P \Rightarrow 0P0 \Rightarrow 01P10 \Rightarrow 0110$ and we have inferred that $0110 \in L_{pal}$.

We denote that there is a derivation that requires zero or more derivation steps with $\stackrel{*}{\Rightarrow}$ symbol. For example, to indicate that there is a derivation of string 0110 from variable P using some number of steps, is denoted $P \stackrel{*}{\Rightarrow} 0110$.

3.2.2 Parse Trees for a Grammar

There is a tree representation for derivations that show explicitly how terminal symbol are grouped into substrings, each of which belongs to the language of one of the variables of the grammar. These trees are called *parse trees*. There might be more than one parse tree for a terminal string that belongs to the language of some grammar. In that case the grammar is called *ambiguous*. Ambiguous grammars are not suitable for representing a syntax of a programming language unless the ambiguities are resolved somehow.

The parse trees of a specific grammar G are trees with the following conditions:

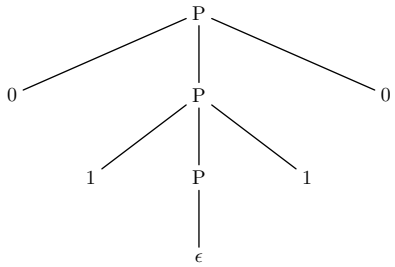
1. Each interior node is labeled by a variable of the grammar.
2. Each leaf is labeled by either a variable, a terminal, or ϵ. However, if the leaf is labeled ϵ, then it must be the only child of its parent.
3. If an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

respectively, from the left, then $A \to X_1X_2 \cdots X_k$ is a production of the grammar G.

Figure 3.1 shows a parse tree of derivation $P \stackrel{*}{\Rightarrow} 0110$ for the grammar 3.1.1.

Figure 3.1: A parse tree for derivation $P \stackrel{*}{\Rightarrow} 0110$



3.2.3 Compact Notation for Grammars

Let $\omega_1, \omega_2, \ldots, \omega_k$ be strings of grammar symbols (i.e. strings of terminals and nonterminals). If we have productions

$$
\begin{aligned}
P &\to \omega_1 \\
P &\to \omega_2 \\
&\cdots \\
P &\to \omega_k
\end{aligned}
$$

in some grammar G, we may represent the P-productions (i.e. the productions whose head is P) by grouping them together as follows:

$P \to \omega_1 \,|\, \omega_2 \,|\, \cdots \,|\, \omega_k$

For example, the grammar 3.1.1 may be represented more compactly as

$$P \to \epsilon \,|\, 0 \,|\, 1 \,|\, 0P0 \,|\, 1P1$$

3.3 Syntax-Directed Translation

Consider the following grammar:

Grammar 3.3.1.

$$\begin{aligned} expr &\rightarrow expr + term \,|\, expr - term \,|\, term \\ term &\rightarrow term * factor \,|\, term/factor \,|\, factor \\ factor &\rightarrow 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9 \,|\, (\, expr\,) \end{aligned}$$

The language defined by this grammar consists of expressions that are lists of terms separated by operator symbols + and −. Terms are in turn lists of factors separated by operator symbols ∗ and /. Factors consist of single digits and parenthesized expressions. The alphabet of this language is $\{+, -, *, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,)\}$.

To see that an expression "1+3*(4-2)", for example, is in this language, we may construct a derivation for it:

$$\begin{aligned} expr &\Rightarrow expr + term \\ &\Rightarrow term + term \\ &\Rightarrow factor + term \\ &\Rightarrow 1 + term \\ &\Rightarrow 1 + term * factor \\ &\Rightarrow 1 + factor * factor \\ &\Rightarrow 1 + 3 * factor \\ &\Rightarrow 1 + 3 * (expr) \\ &\Rightarrow 1 + 3 * (expr - term) \\ &\Rightarrow 1 + 3 * (term - term) \\ &\Rightarrow 1 + 3 * (factor - term) \\ &\Rightarrow 1 + 3 * (4 - term) \\ &\Rightarrow 1 + 3 * (4 - factor) \\ &\Rightarrow 1 + 3 * (4 - 2) \end{aligned}$$

Suppose now that we need to translate infix expressions of this kind into *postfix notation*. The postfix notation of an expression E can be defined inductively as follows:

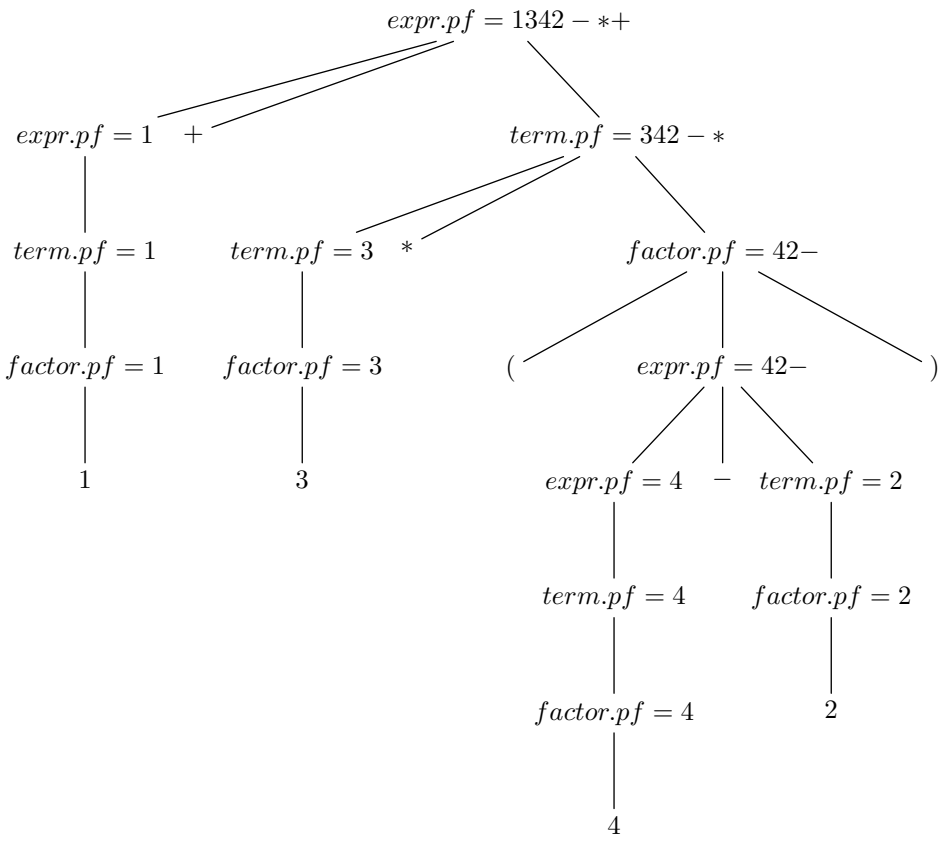
1. If E is a digit, the postfix notation of E is E itself.
2. If E is of the form $E_1 + E_2$, the postfix notation of E is the postfix notation of E_1 followed by the postfix notation of E_2 followed by +.
3. If E is of the form $E_1 * E_2$, the postfix notation of E is the postfix notation of E_1 followed by the postfix notation of E_2 followed by *.
4. If E is of the form (E), the postfix notation of (E) is the postfix notation of E.

For example, postfix notation for infix expression "1+3*(4-2)" is "1342-*+".

In computing the postfix notation from infix expressions, we can take advantage of the grammar 3.3.1 by associating *attributes* to each nonterminal of the grammar. Attributes can in principle be of any kind: numbers, structures or strings, for example. In this case we may represent the value of a postfix expression with one string attribute. A parse tree that shows the values of the attributes of nonterminals is called an *annotated* parse tree.

Figure 3.2 shows an annotated parse tree with an attribute *pf* associated with nonterminals *expr*, *term* and $factor$.

Figure 3.2: Annotated parse tree for expression "1+3*(4-2)"



There can be two kinds of attributes for nonterminals: ([1] pg. 304)

1. A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic action associated with the production at N. A synthesized attribute at node N is defined in terms of attribute values at the children of N and at N itself. The *pf* attribute in Fig. 3.2 is an example of a synthesized attribute.

2. An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic action associated with the production at the *parent* of N. An inherited attribute at node N is defined in terms of attribute values at N's parent, N itself, and N's siblings.

The attributes can be computed by visiting the nodes of the parse tree in some order. Synthesized attributes have the nice property that their values can be computed by a single bottom-up travelsal of the parse tree.

3.4 Parsing

Parsing is the process of determining how a string of terminals can be generated by a grammar. ([1] pg. 60). Most parsing methods fall into one of two classes, called the *top-down* and *bottom-up* methods. These terms refer to the order in which nodes in the parse tree are constructed. In top-down parsers, construction starts at the root and proceeds towards the leaves, while in bottom-up parsers, construction starts the the leaves and proceeds towards the root. Most handwritten parsers use top-down methods, while many parser-generator tools generate a bottom-up parser.

3.4.1 Recursive Descent Parsing

A *recursive-descent parsing* is a top-down method in which a set of recursive procedures is used to process the input. For example, consider the following grammar:

Grammar 3.4.1.

$$stmt \rightarrow \textbf{if}\,(\,expr\,)\,stmt\,\textbf{else}\,stmt$$

To write a recursive-descent parser for this grammar, one writes a procedure that is used to match tokens and obtain more input, and then a procedure for each nonterminal. The following listing shows the structure of these procedures:

```
1  int lookahead;
2
3  void match(int token)
4  {
5      if (token == lookahead)
6      {
7          // read next token into lookahead;
8      }
9      else
10     {
11         throw std::runtime_error("syntax error");
12     }
13 }
14
15 void expr()
16 {
17     // match an expression...
18 }
19
20 void stmt()
21 {
22     match(IF); match('('); expr(); match(')'); stmt(); match(ELSE); stmt
           ();
23 }
```

3.4.2 Left Recursion

A recursive-descent parser cannot directly use grammars like the grammar 3.3.1, because it has "left-recursive" productions such as $expr \rightarrow expr{+}term$, where the leftmost symbol of the body is the same as the nonterminal at the head of the production. Suppose the procedure for *expr* decides to apply this production. The body begins with *expr* so the procedure for *expr* is called recursively. Since the lookahead symbol changes only when a terminal is matched, no change to the input took place between recursive calls of *expr*. As a result, the second call to *expr* does exactly what the first call did, which means a third call, and so on.

A left-recursive production can be eliminated by rewriting the offending production. Consider a nonterminal A with two productions

$$A \rightarrow A\alpha \,|\, \beta$$

where α and β are sequences of terminals and nonterminals that do not start with A. For example, in

$$expr \rightarrow expr + term \,|\, term$$

nonterminal $A = expr$, string $\alpha = +term$, and string $\beta = term$.

The nonterminal A and its production are said to be *left recursive* ([1] pg. 67), because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol of the right side. Repeated application of this production builds up a sequence of α's to the right of A. When A is finally replaced by β, we have a β followed by a sequence of zero or more α's.

We can achieve the same effect by rewriting the productions for A in the following manner, using a new nonterminal R:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \,|\, \epsilon \end{aligned}$$

3.5 Extending the Grammar Notation

We have found it useful to extend the context-free grammar notation with regular-expression like operations. [1] [2] In the following definitions the expression in the middle is in the extended form, and the productions on the right express the same language using conventional context-free grammar notation.

1. X or Y:

$$P \rightarrow \alpha\,(X \,|\, Y)\,\beta \qquad\qquad \begin{aligned} P &\rightarrow \alpha R \beta \\ R &\rightarrow X \,|\, Y \end{aligned}$$

2. Closure of X, X occurs zero or more times:

$$P \rightarrow \alpha X^* \beta \qquad\qquad \begin{aligned} P &\rightarrow \alpha R \beta \\ R &\rightarrow RR \,|\, X \,|\, \epsilon \end{aligned}$$

[1] α and β denote strings of grammar symbols, and X and Y single grammar symbols.

[2] Since asterisk, plus, question mark, parentheses and square brackets belong to regular expression syntax, they must now be quoted when they appear as terminals in productions of extended notation.

3. Positive X, X occurs one or more times:

$$P \to \alpha X^{+} \beta \qquad\qquad P \to \alpha R \beta \quad R \to RR \,|\, X$$

4. Optional X, X occurs zero or one times:

$$P \to \alpha X ? \beta \qquad\qquad P \to \alpha R \beta \quad R \to X \,|\, \epsilon$$

5. Class [abc], one of the characters in the class occurs:

$$P \to \alpha \,[abc]\, \beta \qquad\qquad P \to \alpha R \beta \quad R \to a \,|\, b \,|\, c$$

In the definitions above, X denotes a single grammar symbol, i.e. either terminal or nonterminal, but we may extend the notation further by substituting X with arbitrary expressions containing grammar symbols and other expressions, much the same way we can use regular expressions. We can now replace left recursion with iteration using the extended notation. The left-recursive productions

$$A \to A\alpha \,|\, \beta$$

become an iterative production:

$$A \to \beta(\alpha)^{*}$$

meaning β followed by zero or more α's.

We can rewrite the grammar 3.3.1 without left recursion using the extended notation as follows:

Grammar 3.5.1.

$$\begin{aligned} expr &\to term\ (\ ('+'|'-')\ term\ )^{*} \\ term &\to factor\ (\ ('*'|'/')\ factor\ )^{*} \\ factor &\to [0-9] \,|\, '('\ expr\ ')' \end{aligned}$$

3.6 Parsing in Cmajor

The parsers in Cmajor are written using the Cmajor Parser Generator, or `cmpg`, notation, that is much like the extended grammar notation of the previous section. The `cmpg` reads grammar definitions in *.parser* files, validates them, and generates C++ classes that represent the grammars. To become familiar with the grammar definition syntax, we write the grammar 3.5.1 using the `cmpg` notation.

Example 3.6.1. Postfix Translation Grammar.

```
1  grammar PostfixTranslationGrammar
2  {
3      expr: std::string
4          ::= term:t{ value = t; }
5          (   '+' term:pt{ value.append(pt).append(1, '+'); }
6          |   '-' term:mt{ value.append(mt).append(1, '-'); }
7          )*
8          ;
9
10     term: std::string
11         ::= factor:f{ value = f; }
12         (   '*' factor:tf{ value.append(tf).append(1, '*'); }
13         |   '/' factor:df{ value.append(df).append(1, '/'); }
14         )*
15         ;
16
17     factor: std::string
18         ::= digit{ value = std::string(1, *matchBegin); }
19         |   '(' expr{ value = expr; } ')'
20         ;
21 }
```

The grammar has a list of *rules*. In this case *expr*, *term* and *factor*. If the start rule is not explicitly defined by the **start** clause, the first rule of the grammar is taken as the start rule.

A rule may have one synthesized attribute whose type is denoted by a colon and a name of a C++ type after the head of the rule, `std::string` in this case. In this example each of the rules of the grammar have a synthesized attribute of type `std::string`. If multiple synthesized attributes are needed, one can specify a structure of values, or a dynamically created object holding the values.

The ::= symbol corresponds to the $\rightarrow$ symbol in the formal grammars.

If the same nonterminal occurs many times inside the body of a rule, and that nonterminal refers to a rule that has a synthesized attribute, the synthesized attribute has to be named explicitly by a colon and an identifier after the name of the nonterminal. In the body of the *expr* rule, for example, one can refer to many occurrences of *term*'s synthesized attribute, the first of which is named *t*, the second *pt*, and the third *mt*.

A grammar symbol in a body of a rule may have an associated semantic action, i.e. a block of C++ code. For example in line 4, the first *term* nonterminal has a semantic action `{ value = t; }` associated with it. The semantic action is executed only if input matches the rule that it is associated with.

The synthesized attribute of the rule is exposed as an identifier *value* inside the body of a rule. It can be read and assigned to many times inside the body of a rule. For example in line 4, the value of the synthesized attribute of the *expr* rule is initialized to a value of the synthesized attribute of the *term* rule. When more *term*s are matched, the synthesized attributes of these are appended to the synthesized attribute the *expr* rule.

The matched lexeme of a grammar symbol is exposed as two character pointers to the semantic action associated with a grammar symbol. The *matchBegin* pointer points to the start of the matched lexeme and the *matchEnd* pointer points to one past the end of the

matched lexeme. For example, in line 18, the value of the matched digit is assigned to the synthesized attribute of the *factor* rule.

If the nonterminal occurs only once inside the body of a rule, one can refer the synthesized attribute of it with the name of the nonterminal. Example of this appears in the line 19, where the synthesized attribute of *expr* rule is referred in the semantic action by its name *expr*.

3.6.1 Internal Representation of `cmpg` Grammar Definitions

The `cmpg` program reads grammar definitions and constructs an internal representation for them. The internal representation of a grammar is a list of rules, one of which is set as a start rule. Each rule has a *name* and a *definition*. The definition of a rule is represented as a *tree of parsing nodes*.

There are many kinds of parsing nodes. Each kind of parsing node has either zero, one, or two child nodes. A node that has zero child nodes is also called a *leaf* parsing node, a node that has one child node is called a *unary* parsing node, and a node that has two child nodes is called a *binary* parsing node.

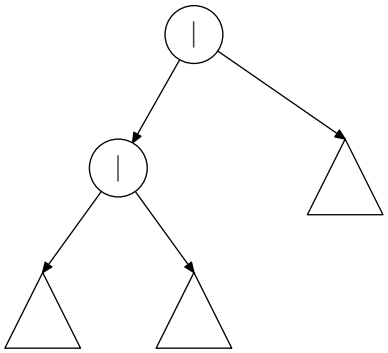
- The definition of a rule consists of nonempty sequence of *alternative* expressions:

$$R \to \omega_1 \,|\, \omega_2 \,|\, \cdots \,|\, \omega_k$$

  If input matches one of the alternatives, it matches the rule. The alternatives are tested from left to right, and if a match is found, the rest of the alternatives are not tested.

  If the definition of a rule is represented as a tree of parsing nodes, it consists of *alternative* binary parsing nodes, where the left and right subtrees of an alternative nodes represent expressions ω_i and ω_{i+1}. Figure 3.3 shows two alternative nodes.

Figure 3.3: Alternative Nodes



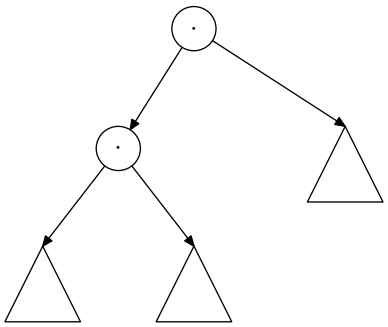
- Each alternative expression ω_i consists of catenation of expressions :

$$\alpha_1 \alpha_2 \cdots \alpha_k$$

If input consists of a nonempty sequence of strings $s_1, s_2, \ldots, s_k$ of terminal symbols where s_1 matches expression α_1, s_2 matches expression α_2, etc., and s_k matches expression α_k, the input matches the whole alternative expression.

A *catenate* node is a binary parsing node, whose left and right subtree represent expressions α_i and α_{i+1}. Figure 3.4 shows two catenate nodes.

Figure 3.4: Catenate Nodes



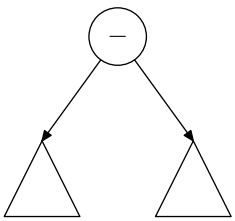
- A *difference* expression is denoted by α_i in a catenate expression $\alpha_1\alpha_2\cdots\alpha_k$. The difference expression consists of nonempty sequence of expressions separated by the $-$ symbol:

$$\beta_1 - \beta_2 - \cdots - \beta_k$$

Usually $k = 1$ or $k = 2$. If a string s of terminal symbols matches expression β_1, but does not match expression β_2, the string s matches expression $\beta_1 - \beta_2$.

A *difference* node is a binary parsing node whose left and right subtrees represent expressions β_1 and β_2 respectively. Figure 3.5 shows a difference node.

Figure 3.5: Difference Node



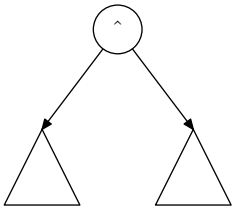
- An *xor* expression is denoted by β_i in a difference expression $\beta_1 - \beta_2 - \cdots - \beta_k$. The xor expression consists of nonempty sequence of expressions separated by the $\hat{}$ symbol:

$$\gamma_1\hat{}\gamma_2\hat{}\cdots\hat{}\gamma_k$$

Usually $k = 1$ or $k = 2$. If a string s of terminal symbols either matches expression γ_1, but does not match expression γ_2, or matches expression γ_2, but does not match expression γ_1, the string s matches expression $\gamma_1\hat{}\gamma_2$.

An *xor* node is a binary parsing node whose left and right subtrees represent expressions $\gamma 1$ and $\gamma 2$ respectively. Figure 3.6 shows an xor node.

Figure 3.6: Xor Node



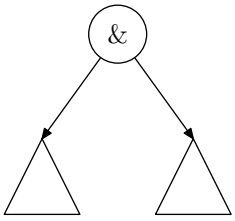
- An *intersection* expression is denoted by γ_i in an xor expression $\gamma_1\hat{}\gamma_2\hat{}\cdots\hat{}\gamma_k$. The intersection expression consists of nonempty sequence of expressions separated by the & symbol:

$$\mu_1 \,\&\, \mu_2 \,\&\, \cdots \,\&\mu_k$$

Usually $k = 1$ or $k = 2$. If a string s of terminal symbols matches both expression μ_1 and expression μ_2, the string s matches expression $\mu_1\&\mu_2$.

An *intersection* node is a binary parsing node whose left and right subtrees represent expressions μ_1 and μ_2 respectively. Figure 3.7 shows an intersection node.

Figure 3.7: Intersection Node



- A *list* expression is denoted by μ_i in an intersection expression $\mu_1 \,\&\, \mu_2 \,\&\, \cdots \,\&\mu_k$: The list expression is an expression optionally followed by the % symbol and an expression:

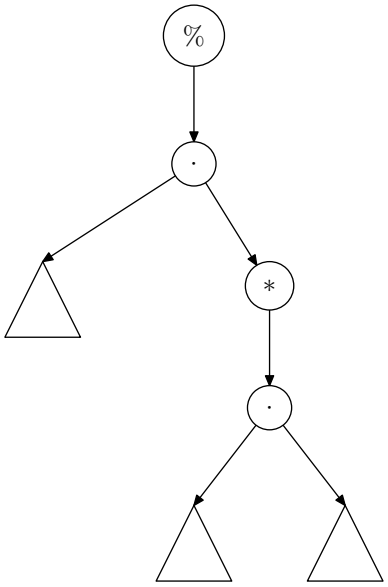
  $$\theta_1 \,(\%\, \theta_2)?$$

  In the previous expression the parentheses and the ? symbol are metasymbols, not terminal symbols.

  Expression $\theta_1 \% \theta_2$ denotes a nonempty sequence of θ_1's separated by θ_2's.

  A list node is a unary parsing node, whose child subtree is set to nodes corresponding to expression $\theta_1(\theta_2\theta_1)^*$. Figure 3.8 shows a list node with a child subtree.

  Figure 3.8: List Node

- A *postfix* expression is denoted by θ_i in a list expression $\theta_1(\,\%\,\theta_2)$? A postfix expression is an expression optionally followed by one of the symbols $*$, $+$, or ?:

  $$\eta('*'|'+'|'?')?$$

  In the previous expression the parentheses and the last ? symbol are metasymbols, not terminal symbols.

  The postfix expressions containing symbols $*$, $+$, and ? are:

  1. η^*: If the input consists of a possibly empty sequence of strings s_i of terminal symbols where each string s_i mathes expression η, the input matches expression η^*. For example, strings $\{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}\}$ match expression $\mathbf{a}^*$.
     A *closure* node is a unary parsing node whose child subtree represents expression η.

2. η^+: If the input consists of a nonempty sequence of strings s_i of terminal symbols where each string s_i mathes expression η, the input matches expression η^+. For example, strings $\{\texttt{a}, \texttt{aa}, \texttt{aaa}\}$ match expression $\mathbf{a}^+$.

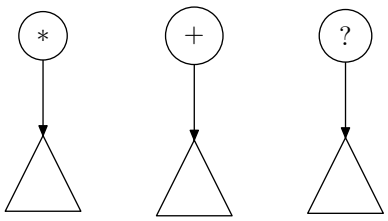
   A *positive* node is a unary parsing node whose child subtree represents expression η.

3. $\eta?$: If the input consists either an empty string ϵ, or a string s of terminal symbols where s matches expression η, the input matches expression $\eta?$. For example, strings $\{\epsilon, \texttt{a}\}$ match expression $\mathbf{a}?$.

   An *optional* node is a unary parsing node whose child subtree represents expression η.

Figure 3.9 shows the postfix nodes.

Figure 3.9: Postfix Nodes



- A *primary* expression is denoted by η in a postfix expression $\eta('*'|'+'|'?')?$.

  Using extended context-free grammar notation, a primary expression can be expressed as:

$$primary \rightarrow (\ primitive \mid nonterminal \mid grouping \mid token\ )\ expectation?\ action?$$

  That is, a primary expression is one of:

  1. a *primitive* expression, that is an atomic `cmpg` expression.
  2. a *nonterminal* expression that matches input to a rule recursively.
  3. a *grouping* expressions that is a parenthesized alternative expression.
  4. a *token* expression that prevents skipping.

  Previous expressions can be optionally followed by an *expectation* expression that prevents backtracking, and an *action* expression that associates a semantic action to a primary expression.

- The primitive expression is defined using the extended context-free notation as:

$$\begin{aligned} primitive \rightarrow\ & char \mid string \mid charset \mid keyword \mid keyword\_list \mid \\ & \textbf{empty} \mid \textbf{space} \mid \textbf{anychar} \mid \textbf{letter} \mid \textbf{digit} \mid \textbf{hexdigit} \mid \textbf{punctuation} \end{aligned}$$

Figure 3.10 shows the primitive expressions, what input they match, and the corresponding node types.

Figure 3.10: Primite Expressions

Expression	Matches	Node
char	matches a single terminal symbol to a character specified in the expression.	'x'
string	matches a string of terminal symbols to a string specified in the expression.	"abc"
charset	matches a single terminal symbol to set of characters specified in the expression.	[abc]
keyword	matches a string of terminal symbols to a keyword string specified in the expression.	**for**
keyword_list	matches a string of terminal symbols to a list of keyword strings specified in the expression	**for,if**
empty	matches always	**empty**
space	matches a single terminal symbol to any whitespace character	**space**
anychar	matches a single terminal symbol to any single character	**anychar**
letter	matches a single terminal symbol to any latin letter	**letter**
digit	matches a single terminal symbol to any decimal digit	**digit**
hexdigit	matches a single terminal symbol to any hexadecimal digit	**hexdigit**
punctuation	matches a single terminal symbol any ASCII punctuation symbol	**punct**

- A *nonterminal* expression is defined using extended context-free notation as follows:

$$\begin{aligned} nonterminal &\rightarrow (\ identifier \mid identifier\ arguments\ )\ alias? \\ arguments &\rightarrow\ '('argument(\ ','\ argument\ )^*\ ')' \\ alias &\rightarrow\ ':'\ identifier \end{aligned}$$

The nonterminal expression names a rule that is matched recursively. It can contain a parenthesized list of *arguments*, that become the inherited attributes of the "called" rule. We used the word "called" because the recursive matching process can be thought as procedures that call each other recursively, as in recursive-descent parser.

If the called rule has a synthersized attribute and the rule is called many times inside a body of a rule, the synthesized attribute of the called rule must be given a unique name. That is the use of an *alias* expression.

The node for the nonterminal is represented as

$$\boxed{nt(foo)}$$

where *foo* is the name of the rule matched recursively.

- A *grouping* expression is a parenthesized sequence of alternative expressions.

$$grouping \rightarrow \,'('\, alternatives\,')'$$

- A *token* expression consists of a keyword **token** followed by a parenthesized sequence of alternative expressions. It prevents skipping of tokens that match the *skip rule* of the grammar.

$$token \rightarrow \textbf{token}\,'('\, alternatives\,')'$$

- An *expectation* expression is a single '!' symbol associated with the preceding primary expression. It forces the matching of its preceding expression without backtracking. If its associated expression does not match, an exception is thrown.

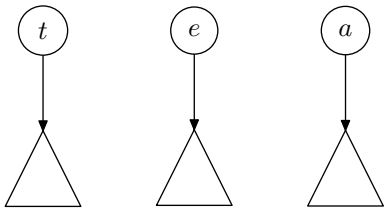
$$expectation \rightarrow \,'!'$$

- An *action* expression is a block of C++ code in braces. It represents a semantic action that is executed if input matches its associated primary expression.

$$action \rightarrow \,'\{'\,\texttt{C++ code}\,'\}'$$

Figure 3.11 shows the token, expectation and action unary parsing nodes.

Figure 3.11: Token, Expectation, and Action Nodes



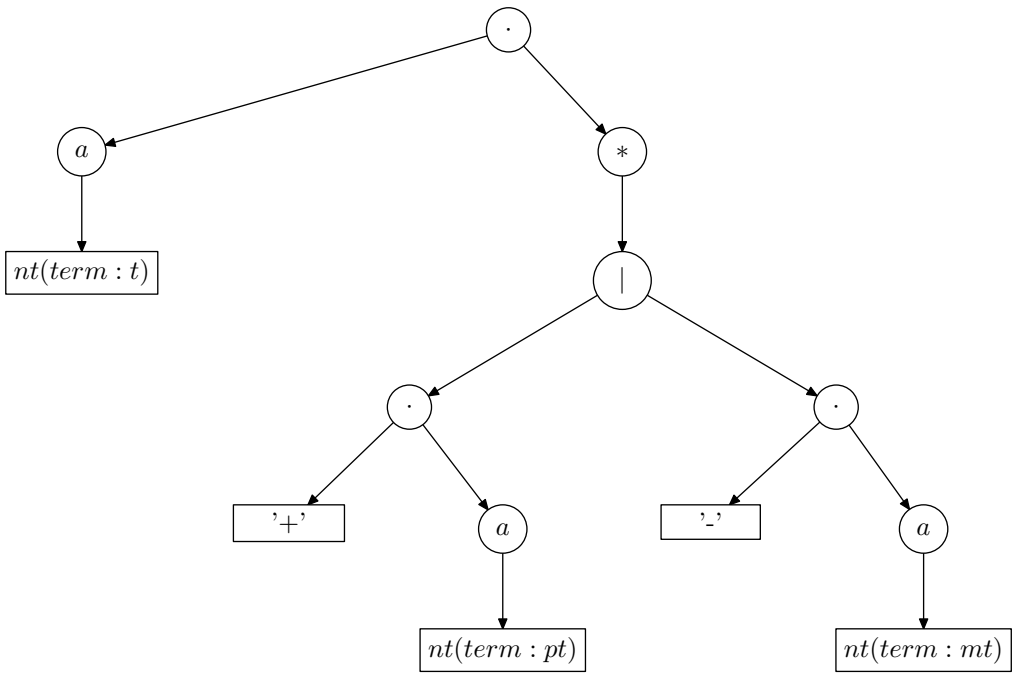
Example 3.6.2. Example of Internal Representation.

Let us recall the Postfix Translation Grammar of example 3.6.1. For ease of reference it is repeated here:

```
1  grammar PostfixTranslationGrammar
2  {
3      expr: std::string
4          ::= term:t{ value = t; }
5          (   '+' term:pt{ value.append(pt).append(1, '+'); }
6          |   '-' term:mt{ value.append(mt).append(1, '-'); }
7          )*
8          ;
9
10     term: std::string
11         ::= factor:f{ value = f; }
12         (   '*' factor:tf{ value.append(tf).append(1, '*'); }
13         |   '/' factor:df{ value.append(df).append(1, '/'); }
14         )*
15         ;
16
17     factor: std::string
18         ::= digit{ value = std::string(1, *matchBegin); }
19         |   '(' expr{ value = expr; } ')'
20         ;
21 }
```

Figure 3.12 shows the internal representation of the *expr* rule.

Figure 3.12: Internal Representation of *expr* Rule



3.6.2 `cmpg` Language Grammar

Here the syntax of the `cmpg` language is presented in extended context-free notation:

Grammar 3.6.1. `cmpg` Language Grammar.

$$
\begin{aligned}
grammar &\rightarrow \textbf{grammar}\ identifier\ '\{'\ grammarcontent\ '\}' \\
grammarcontent &\rightarrow (\ startclause \mid skipclause \mid rulelink \mid rule\ )^* \\
startclause &\rightarrow \textbf{start}\ identifier\ ';' \\
skipclause &\rightarrow \textbf{skip}\ qualifiedid\ ';' \\
rulelink &\rightarrow \textbf{using}\ (\ identifier\ '='\ qualifiedid \mid qualifiedid\ )\ ';' \\
rule &\rightarrow identifier\ locals?\ returns?\ "\ ::=\ "\ alternatives\ ';' \\
locals &\rightarrow '('\ (variable \mid parameter)\ (\ ','\ (variable \mid parameter))^*\ ')' \\
variable &\rightarrow \textbf{var}\ cpptype\ cppdeclarator \\
parameter &\rightarrow cpptype\ cppdeclarator \\
returns &\rightarrow\ ':'\ cpptype \\
alternatives &\rightarrow catenate\ (\ '|'\ catenate\ )^* \\
catenate &\rightarrow diff^+ \\
diff &\rightarrow xor\ (\ '-'\ xor)^* \\
xor &\rightarrow and\ (\hat{}and)^* \\
and &\rightarrow list\ (\ '\&'\ list)^* \\
list &\rightarrow postfix\ (\ '\%'list)? \\
postfix &\rightarrow primary\ (\ '*'\ \mid '+'\ \mid '?'\ )? \\
primary &\rightarrow (\ primitive \mid nonterminal \mid grouping \mid token\ )\ expectation?\ action? \\
primitive &\rightarrow\ char \mid string \mid charset \mid keyword \mid keyword\_list \\
&\quad \mid \textbf{empty} \mid \textbf{space} \mid \textbf{anychar} \mid \textbf{letter} \mid \textbf{digit} \mid \textbf{hexdigit} \mid \textbf{punctuation} \\
nonterminal &\rightarrow (\ identifier \mid identifier\ arguments\ )\ alias? \\
arguments &\rightarrow\ '('argument(\ ','\ argument\ )^*\ ')' \\
alias &\rightarrow\ ':'\ identifier \\
grouping &\rightarrow\ '('\ alternatives\ ')' \\
token &\rightarrow \textbf{token}\ '('alternatives\ ')' \\
expectation &\rightarrow\ '!' \\
action &\rightarrow\ '\{'\ \texttt{C++ code}\ '\}' \\
identifier &\rightarrow\ id\ -\ keyword \\
qualifiedid &\rightarrow\ identifier\ (\ '.'\ identifier)^* \\
id &\rightarrow\ (\ \textbf{letter} \mid '\_'\ )\ (\ \textbf{letter} \mid \textbf{digit} \mid '\_'\ )^* \\
keyword &\rightarrow \textbf{using} \mid \textbf{grammar} \mid \textbf{start} \mid \textbf{skip} \mid \textbf{token} \mid \textbf{keyword} \mid \textbf{keyword\_list} \\
&\quad \mid \textbf{empty} \mid \textbf{space} \mid \textbf{anychar} \mid \textbf{letter} \mid \textbf{digit} \mid \textbf{hexdigit} \mid \textbf{punctuation} \mid \textbf{var}
\end{aligned}
$$

The *cpptype* denotes a `C++` type expression, and the *cppdeclarator* denotes a `C++` declarator.

3.6.3 Informal Description of Operation of a Parser Generated Using `cmpg`

A parser generated using `cmpg` works much the same way than a handwritten recursive-descent parser would operate. In principle, each rule can be thought as a recursive procedure that receives parameters, or inherited attributes, from its caller, or parent rule, matches terminals and maybe calls other recursive procedures, or rules, and finally can return a value, a computed synthesized attribute, to its caller, or parent rule.

The parsing begins by trying to match the start of the input to the body of the rule S, the start rule of the grammar.

If the current input position is at the start of rule P, and there are many P-productions, $P \to \omega_1 \,|\, \omega_2 \,|\, \cdots \,|\, \omega_k$, the parser tries to match the the input to the production $P \to \omega_1$. If the input matches, the other P-productions are not tried and the parsing proceeds to the successor of the caller of the production $P \to \omega_1$. However, if the input does not match $P \to \omega_1$, input is backtracked, and the production $P \to \omega_2$ is tried, and so on, until either a match is found, or the input did not match the last P-production $P \to \omega_k$. In that case, let $Q \to \alpha P \beta \Leftrightarrow Q \to \upsilon_i$ be the parent of P. At this point the input is backtracked and the next alternative for the caller of the P, $Q \to \upsilon_{i+1}$ is tried. This process is repeated until either the entire input matches, or a syntax error is detected.

3.6.4 Parsing Algorithm

The algorithm uses a stack of attribute values, a Boolean variable for skipping state *skip*, a stack of skipping states, and keeps track of *current input position*. Each rule has a data structure called *context* that contains the current values of inherited attributes, synthesized attribute, local variables, and synthesized attributes of the contained nonterminals of the rule. Each rule has also a stack of those context structures called a *context stack*.

When input is parsed using the following algorithm 3.6.1 applied to a parsing node, the result of parsing can be either:

1. **match**(**true**, n), where $n > 0$, to indicate that input matched, and the length of the match was n characters.
2. **match**(**true**, 0), to indicate a successful empty match. In this case the current input position was not advanced.
3. **match**(**false**) to indicate that input did not match. In this case we say that the result is a *failure* match.

In the beginning the attribute stack is empty, the skipping state stack is empty, and the skipping state *skip* is **true**. The parsing begins by setting the current input position to the start of the input, and applying algorithm 3.6.1 to the root node of the parsing node tree that forms the definition of the start rule of the grammar. Let m be the result of parsing applied to the root node.

If m is:

1. **match**(**true**, n), where n is the length of the input, the parsing succeeds.
2. **match**(**true**, n), where n is less than the length of the input, the parsing fails.
3. **match**(**false**), the parsing fails.

Algorithm 3.6.1. Parsing Algorithm. ([3])

If the type of the node this algorithm is applied to is:

1. Alternative node (Fig. 3.3). Let *save* be the current input position. Apply this algorithm recursively to the left subtree of this node. Let m be the result of parsing the left subtree. [3] If m was a successful match, let the result of parsing this node be m.

   Otherwise, backtrack by setting the current input position to *save* and apply this algorithm recursively to the right subtree of this node. Let the result of parsing this node be the result of parsing the right subtree.

2. Catenate node (Fig. 3.4). Apply this algorithm recursively to the left subtree of this node. Let m_1 be the result of parsing the left subtree. If m_1 a successful match, unless *skip* is **false** skip tokens using the skip rule, then apply this algorithm recursively to the right subtree of this node. Let m_2 be the result of parsing the right subtree. If m_2 was a successful match, let the result of parsing this node be $\textbf{match}(\textbf{true}, length(m_1) + length(m_2))$.

   Otherwise, either m_1 was a failure match, or m_2 was a failure match. Let the result of parsing this node be **match**(**false**).

3. Difference node (Fig. 3.5). Let *save* be the current input position. Apply this algorithm recursively to the left subtree of this node. Let m_1 be the result of parsing the left subtree. If m_1 was a successful match, let *tmp* be the current input position, and backtrack by setting the current input position to *save*; then apply this algorithm recursively to the right subtree of this node. Let m_2 be the result of parsing the right subtree. If m_2 was a failure match, or $length(m_2) < length(m_1)$, set the current input position to *tmp*, and let the result of parsing this node be m_1, a successful match.

   Otherwise, either m_1 was a failure match, or m_2 was a successful match with $length(m_2) \geq length(m_1)$. Let the result of parsing this node be **match**(**false**).

4. Xor node (Fig. 3.6). Let *save* be the current input position. Apply this algorithm recursively to the left subtree of this node. Let m_1 be the result of parsing the left subtree. Let *tmp* be the current input position, and backtrack by setting the current input position to *save*. Apply this algorithm recursively to the right subtree of this node. Let m_2 be the result of parsing the right subtree. If m_1 was a successful match and m_2 was a failure match, or m_1 was a failure match and m_2 was a successful match, do the following:

   (a) If m_1 was a successful match, set the current input position to *tmp*.

   (b) If m_1 was a successful match, let the result of parsing this node be m_1, otherwise let the result of parsing this node be m_2.

   Otherwise, either both m_1 and m_2 were successful matches, or both were failure matches. Let the result of parsing this node be **match**(**false**).

[3] When we say that a node, or a subtree, is parsed, we mean that input is parsed in the context of that node, or subtree.

5. Intersection node (Fig. 3.7). Let *save* be the current input position. Apply this algorithm recursively to the left subtree of this node. Let m_1 be the result of parsing the left subtree. If m_1 was a successful match, backtrack by setting the current input position to *save*, and apply this algorithm recursively to the right subtree of this node. Let m_2 be the be the result of parsing the right subtree. If m_2 was a successful match and $length(m_1) = length(m_2)$, let the result of parsing this node be m_1.

   Otherwise, either m_1 was a failure match, m_2 was a failure match, or $length(m_1) \neq length(m_2)$. Let the result of parsing this node be **match**(**false**).

6. List node (Fig. 3.8). Apply this algorithm recursively to the child subtree of this node. Let the result of parsing this node be the result of parsing the child subtree.

7. Closure node (Fig. 3.9). Let m_1 be **match**(**true**, 0), and let $first$ be **true**. Do following in a loop until loop exited:

   (a) Let *save* be the current input position.

   (b) If $first =$ **true**, set $first$ to **false**, otherwise, unless *skip* is **false**, skip tokens using the skip rule.

   (c) Apply this algorithm recursively to the child subtree of this node. Let m_2 be the result of parsing the child subtree.

   (d) If m_2 was a successful match, set m_1 to **match**(**true**, $length(m_1) + length(m_2)$), otherwise backtrack by setting the current input position to *save* and exit the loop.

   Let the result of parsing this node be m_1.

8. Positive node (Fig. 3.9). Apply this algorithm recursively to the child subtree of this node. Let m_1 be the result of parsing the child subtree.

   If m_1 was a successful match, do following in a loop until loop exited:

   (a) Let *save* be the current input position.

   (b) If *skip* is **true**, skip tokens using the skip rule.

   (c) Apply this algorithm recursively to the child subtree of this node. Let m_2 be the result of parsing the child subtree.

   (d) If m_2 was a successful match, set m_1 to **match**(**true**, $length(m_1) + length(m_2)$), otherwise backtrack by setting the current input position to *save* and exit the loop.

   Let the result of parsing this node be m_1.

9. Optional node (Fig. 3.9). Let *save* be the current input position. Apply this algorithm recursively to the child subtree of this node. Let m be the result of parsing the child subtree. If m was a successful match, let the result of parsing this node be m.

   Otherwise, backtrack by setting the current input position to *save*. Let the result of parsing this node be **match**(**true**, 0).

10. Char node (Fig. 3.10). If current input position is not at the end of the input, and the character at the current input position is equal to the character contained in this char node, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1).

    Otherwise, either the current input position is at the end of the input, or the character at the current input position is not equal to the character contained in this char node, so let the result of parsing this node be **match**(**false**).

11. String node (Fig. 3.10). Let m be **match**(**true**, 0). Let i be 0. Let n be the length of the string contained in this string node.

    While $i < n$ and the current input position is not at the end of the input and the character at the current input position is equal to the i'th character of the string contained in this string node, do the following:

    (a) Advance the current input position by one character.
    (b) Increment i.
    (c) Set m to **match**(**true**, $length(m) + 1$).

    If $i = n$, let the result of parsing this node be m.

    Otherwise let the result of parsing this node be **match**(**false**).

12. CharSet node (Fig. 3.10). If current input position is not at the end of the input, do the following:

    (a) If the character set is not an inverse set, and the character at the current input position is in the set, or the character set is an inverse set, and the character at the current input position is not in the set, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1)

    Otherwise let the result of parsing this node be **match**(**false**).

13. Keyword node (Fig. 3.10). If the contained keyword string is denoted by k, the keyword node contains following expression converted to a tree of parsing nodes: $k - \mathbf{token}(kc)$, where c is usually expression $(\mathbf{letter}|\mathbf{digit}|'\_'|'.')^+$, but may also be user supplied *continuation rule*. Let the result of parsing this node be the result of parsing the contained tree of nodes.

14. Keyword list node (Fig. 3.10). The keyword list node has a *selector rule*, that is usually $(\mathbf{letter}|'\_')(\mathbf{letter}|\mathbf{digit}|'\_')^*$, but may also supplied by the user. The node has also a set of keyword strings s.

    Let *save* be the current input position. First the input is parsed with the selector rule. Let m be the result of this parsing, and l be the matched lexeme. If m is a successful match, do the following:

    (a) If the lexeme l matches one of the contained keyword strings s, let the result of parsing this node be m, otherwise backtrack by setting the current input position to *save*.

    Otherwise let the result of parsing this node be **match**(**false**).

15. Empty node (Fig. 3.10). Let the result of parsing this node be **match**(**true**, 0).

16. Space node (Fig. 3.10). If the current input position is not at the end of the input, and the character at the current input position is a whitespace character, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1).

    Otherwise let the result of parsing this node be **match**(**false**).

17. AnyChar node (Fig. 3.10). If the current input position is not at the end of the input, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1).

    Otherwise let the result of parsing this node be **match**(**false**).

18. Letter node (Fig. 3.10). If the current input position is not at the end of the input, and the character at the current input position is a latin letter character, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1).

    Otherwise let the result of parsing this node be **match**(**false**).

19. Digit node (Fig. 3.10). If the current input position is not at the end of the input, and the character at the current input position is a decimal digit character, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1).

    Otherwise let the result of parsing this node be **match**(**false**).

20. HexDigit node (Fig. 3.10). If the current input position is not at the end of the input, and the character at the current input position is a hexadecimal digit character, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1).

    Otherwise let the result of parsing this node be **match**(**false**).

21. Punctuation node (Fig. 3.10). If the current input position is not at the end of the input, and the character at the current input position is ASCII punctuation character, advance the current input position by one character, and let the result of parsing this node be **match**(**true**, 1).

    Otherwise let the result of parsing this node be **match**(**false**).

22. Nonterminal node. Let the rule that the nonterminal is associated with be r. Parsing proceeds by parsing the rule r recursively as follows:

    (a) Parsing rule r begins by pushing values of arguments specified in this nonterminal node to the attribute stack. Those arguments will become the inherited attributes of r. Arguments can be current values of inherited attributes, the synthesized attribute, local variables, or synthesized attributes of the contained nonterminals of the current rule, i.e. the rule that contains the current nonterminal node.

    (b) On entry of parsing the rule r, the current context structure of r is pushed to the context stack of r and the context of r is initialized with default values.

(c) Then arguments are popped off from the attribute stack, and placed to the context structure of r as inherited attributes.

(d) Apply this algorithm recursively to the root node of the parsing node tree that forms the definition of the rule r. Let the result of parsing be m.

(e) On exit of parsing the rule r, if m was a successful match, the value of the synthesized attribute of r, if any, is pushed to the attribute stack. Then in any case, the previous context of r is popped off from the context stack of r, and it becomes the current context of r.

(f) If m was a successful match, the synthesized attribute of r, if any, is popped off from the attribute stack and placed to the context structure of the current rule as synthesized attribute of this nonterminal.

(g) Let the result of parsing this node be m.

23. Token node (Fig. 3.11). Push the current skipping state *skip* to the skipping state stack, and set *skip* to **false**. Apply this algorithm recursively to the child subtree of this node. Let m be the result of parsing the child subtree. Pop the previous skipping state off from the skipping state stack, and assign it to *skip*. Let the result of parsing this node be m.

24. Expectation node (Fig. 3.11). Apply this algorithm recursively to the child subtree of this node. Let m be the result of parsing the child subtree. If m was a failure match, throw *ExpectationFailure* exception, otherwise, let m be the result of parsing this node.

25. Action node (Fig. 3.11). Apply this algorithm recursively to the child subtree of this node. Let m be the result of parsing the child subtree. If m was a successful match, do the following:

    (a) Let *matchBegin* be the start of the matched lexeme and *matchEnd* be one past the end of the matched lexeme. Let *pass* be **true**.

    (b) Call the semantic action associated with this action node by passing pointers *matchBegin* and *matchEnd*, and reference to *pass* as arguments.

    (c) If the semantic action set *pass* to **false**, let the result of parsing this node be **match**(**false**).

    Otherwise, m was a failure match, so if this action has an associated failure action, call it.

    In any case, let the result of parsing this node be m.

3.6.5 Grammars for Cmajor Language Elements

Let us take a look at some language elements of Cmajor programming language and how they are represented using `cmpg` grammars.

3.6.5.1 Basic Types

The grammar for parsing names of basic types is one of the simplest. It consists of an alternative for each keyword of a basic type. The semantic action associated with a keyword of the type creates an abstract syntax tree node for it and assigns it to the synthesized attribute of the rule, that is exposed to semantic actions as an identifier *value*:

```
1  grammar BasicTypeGrammar
2  {
3      BasicType: Cm::Ast::Node*
4          ::=  keyword("bool"){ value = new Cm::Ast::BoolNode(span); }
5          |    keyword("sbyte"){ value = new Cm::Ast::SByteNode(span); }
6          |    keyword("byte"){ value = new Cm::Ast::ByteNode(span); }
7          |    keyword("short"){ value = new Cm::Ast::ShortNode(span); }
8          |    keyword("ushort"){ value = new Cm::Ast::UShortNode(span); }
9          |    keyword("int"){ value = new Cm::Ast::IntNode(span); }
10         |    keyword("uint"){ value = new Cm::Ast::UIntNode(span); }
11         |    keyword("long"){ value = new Cm::Ast::LongNode(span); }
12         |    keyword("ulong"){ value = new Cm::Ast::ULongNode(span); }
13         |    keyword("float"){ value = new Cm::Ast::FloatNode(span); }
14         |    keyword("double"){ value = new Cm::Ast::DoubleNode(span); }
15         |    keyword("char"){ value = new Cm::Ast::CharNode(span); }
16         |    keyword("wchar"){ value = new Cm::Ast::WCharNode(span); }
17         |    keyword("uchar"){ value = new Cm::Ast::UCharNode(span); }
18         |    keyword("void"){ value = new Cm::Ast::VoidNode(span); }
19         ;
20 }
```

span is a name for a structure exposed to semantic actions that represents a range of input positions. It contains four integer attributes:

1. *fileIndex* is an opaque integer given by user in the main parsing function that identifies the file being parsed.
2. *lineNumber* is the line number of the matched lexeme counted from the start of the file being parsed.
3. *start* is the starting position of the matched lexeme.
4. *end* is the ending position of the matched lexeme.

The start and end positions are measured from the beginning of the whole input string given in the main parsing function.

3.6.5.2 Type Expressions

Next we go through the composition of type expressions. In the beginning of type expression grammar there are declarations that begin with the keyword **using**. They are *rule link*s. A rule link refers to a rule defined in another grammar. It brings the name of a rule to the scope of the grammar being defined.

```
1  grammar TypeExprGrammar
2  {
3      using BasicTypeGrammar.BasicType;
4      using IdentifierGrammar.Identifier;
5      using IdentifierGrammar.QualifiedId;
6      using TemplateGrammar.TemplateId;
7      using ExpressionGrammar.Expression;
8      ...
```

The *TypeExpr* rule is the start rule of the *TypeExprGrammar* grammar:

```
1      ...
2      TypeExpr(
3          ParsingContext* ctx,
4          var std::unique_ptr<Cm::Ast::DerivedTypeExprNode> node
5          ): Cm::Ast::Node*
6          ::= empty
7          {
8              ctx->BeginParsingTypeExpr();
9              node.reset(new Cm::Ast::DerivedTypeExprNode(span));
10         }
11             PrefixTypeExpr(ctx, node.get())
12         {
13             node->GetSpan().SetEnd(span.End());
14             value = Cm::Ast::MakeTypeExprNode(node.release());
15             ctx->EndParsingTypeExpr();
16         }
17         /
18         {
19             ctx->EndParsingTypeExpr();
20         }
21     ;
22     ...
```

The *TypeExpr* rule has one inherited attribute, `ctx`, of type `ParsingContext*`, and one local variable, `node`, of type `std::unique_ptr<DerivedTypeExprNode>`.

The body of the rule begins with keyword **empty** that matches anything without consuming any input. The semantic action associated with it constructs an abstract syntax tree node *DerivedTypeExprNode*, that eventually becomes the synthesized attribute of this rule, if the rule happens to match. The reason that the type of `node` is a unique pointer and not an ordinary one is that we don't want to leak memory in the case that the rule does not match.

The type of the inherited attribute `ctx*`, *ParsingContext*, is a class that is used throughout parsing. It contains Boolean flags that guide the parsing, stacks of Boolean flags that hold the previous values of those flags, and member functions for manipulating those flags.

For example, member function *BeginParsingTypeExpr()* pushes the old value of *parsingTypeExpr* flag to the stack and sets the *parsingTypeExpr* flag to **true**. Correspondingly the *EndParsingTypeExpr()* member function pops the previous value of the *parsingTypeExpr* flag off from the stack and assign it to *parsingTypeExpr*. The reason that the flags are manipulated using stacks is that parsing is a highly recursive process, and we may have several instances of the same rule active at one time. Therefore we must push the old value to the stack when we start parsing a rule, and pop it off when we end parsing that rule.

In line 11 we match the *PrefixTypeExpr* rule recursively. We pass `ctx` and pointer to `node` as arguments to the *PrefixTypeExpr* rule. They become inherited attributes of that rule.

The semantic action associated with the *PrefixTypeExpr* nonterminal sets the value of the synthesized attribute of the rule. If the type expression is a simple one, *value* actually receives the simple type expression node contained by *DerivedTypeExprNode*, otherwise value receives the full *DerivedTypeExprNode*.

The semantic action after the `/` symbol starting line 18 is a *failure action*. It is executed if matching the rule fails. Thus we call *BeginParsingTypeExpr()* function at the start of the rule, and *EndParsingTypeExpr()* function at the end of the rule regardless whether matching the rule succeeds or fails.

The next rule of the *TypeExprGrammar* grammar is the *PrefixTypeExpr* rule:

```
1       ...
2       PrefixTypeExpr(
3           ParsingContext* ctx, Cm::Ast::DerivedTypeExprNode* node)
4           ::= keyword("const"){ node->AddConst(); }
5               PostfixTypeExpr(ctx, node):c
6           |   PostfixTypeExpr(ctx, node)
7           ;
8       ...
```

A *prefix* type expression is a *postfix* type expression optionally prefixed by the keyword **const**. It has two inherited attributes, a *parsing context* and a pointer to the abstract syntax tree node we are constructing.

A *postfix* type expression is a *primary* type expression followed by zero or more *postfix type operators* .,&&,&,*, and[]:

```
1       ...
2       PostfixTypeExpr(
3           ParsingContext* ctx, Cm::Ast::DerivedTypeExprNode* node,
4           var Span s)
5           ::= PrimaryTypeExpr(ctx, node){ s = span; }
6           (   '.' Identifier!{ ... }
7           |   "&&"{ node->AddRvalueRef(); }
8           |    '&'{ node->AddReference(); }
9           |    '*'{ node->AddPointer(); }
10          |   '['{ node->AddArray(); }
11              Expression(ctx):dim{ node->AddArrayDimensionNode(dim); }
12              ']'
13          )*
14          ;
15      ...
```

A *primary* type expression is either a name of a basic type, i.e. **bool**, **sbyte**, etc., a template identifier such as *foo*<**int**>, a name of a type, *Symbol* for instance, or a parenthesized *prefix* type expression.

```
    ...
    PrimaryTypeExpr(
        ParsingContext* ctx, Cm::Ast::DerivedTypeExprNode* node)
        ::= BasicType{ node->SetBaseTypeExpr(BasicType); }
        |   TemplateId(ctx){ node->SetBaseTypeExpr(TemplateId); }
        |   Identifier{ node->SetBaseTypeExpr(Identifier); }
        |   '('{ node->AddLeftParen(); } PrefixTypeExpr(ctx, node)! ')'{
            node->AddRightParen(); }
        ;
}
```

3.6.5.3 Template Identifiers

The *template identifier* has one inherited attribute: `ctx` of type `ParsingContext*`, and one local variable `templateId` of type `std::unique_ptr<TemplateIdNode>` that becomes the value of the inherited attribute of the rule.

```
grammar TemplateGrammar
{
    using IdentifierGrammar.Identifier;
    using IdentifierGrammar.QualifiedId;
    using TypeExprGrammar.TypeExpr;

    TemplateId(ParsingContext* ctx,
        var std::unique_ptr<TemplateIdNode> templateId): Cm::Ast::Node*
        ::= empty{ ctx->BeginParsingTemplateId(); }
        (
            QualifiedId:subject
            {
                templateId.reset(new TemplateIdNode(span, subject));
            }
            '<'
            (   TypeExpr(ctx):templateArg
                {
                    templateId->AddTemplateArgument(templateArg);
                }
                %
                ','
            )
            '>'
        )
        {
            ctx->EndParsingTemplateId();
            value = templateId.release();
            value->GetSpan().SetEnd(span.End());
        }
        ...
```

At the beginning of the rule *BeginParsingTemplateId()* member function of the *ParsingContext* is called. Correspondingly at the end of the rule *EndParsingTemplateId()* member function of the *ParsingContext* is called regardless whether the parsing succeeds or fails. *BeginParsingTemplateId()* function pushes the value of member variable *parsingTemplateId* to the stack and sets *parsingTemplateId* to **true**. *EndParsingTemplateId()* function pops the previous value of member variable *parsingTemplateId* off from the stack and assigns it to *parsingTemplateId*.

Template identifier consists of a qualified identifier, *foo*, *bar.bazz*, etc., followed a list of one or more *type expressions* between angle brackets. Thus the *TypeExpr* rule is called recursively by this rule.

```
1           ...
2           /
3           {
4               ctx->EndParsingTemplateId();
5           }
6       ;
7       ...
```

3.6.5.4 Expressions

In the beginning of *Expression* grammar there are some rule link declarations. These are the external rules that this grammar uses:

```
1  grammar ExpressionGrammar
2  {
3      using LiteralGrammar.Literal;
4      using BasicTypeGrammar.BasicType;
5      using IdentifierGrammar.Identifier;
6      using IdentifierGrammar.QualifiedId;
7      using TemplateGrammar.TemplateId;
8      using TypeExprGrammar.TypeExpr;
9      ...
```

The start rule of the grammar is the *Expression* rule. It has one inherited attribute `ctx` of type `ParsingContext`.

An *expression* consists of an *equivalence expression*. The value of `ctx` is passed as an argument to the *Equivalence* rule. After matching *Equivalence*, the synthesized attribute of the *Expression* rule is set to the value of the synthesized attribute of the *Equivalence* rule.

```
1      ...
2      Expression(ParsingContext* ctx): Cm::Ast::Node*
3          ::= Equivalence(ctx){ value = Equivalence; }
4          ;
5      ...
```

An *equivalence* expression consists of a nonempty sequence of *implication* expressions separated by <=> symbols: $\alpha_1 <=> \alpha_2 <=> \cdots <=> \alpha_k$. If $k > 1$ and we are not parsing a concept definition, or we are parsing a template identifier, we reject the input by setting *pass* to **false**. This is the way to make semantic decisions during parsing. An expression of the form $\alpha_1 <=> \alpha_2$ is accepted only in a concept definition. Sole *implication* expression α_1 is accepted always.

```
1       ...
2       Equivalence(ParsingContext* ctx,
3           var std::unique_ptr<Node> expr,
4           var Span s): Cm::Ast::Node*
5           ::=
6           (   Implication(ctx):left{ expr.reset(left); s = span; }
7               (   "<=>"
8                   {
9                       if (!ctx->ParsingConcept()
10                      || ctx->ParsingTemplateId())
11                          pass = false;
12                  }
13                  Implication(ctx):right!
14                  {
15                      s.SetEnd(span.End());
16                      expr.reset(new EquivalenceNode(s, expr.release(),
                            right));
17                  }
18              )*
19          )
20          {
21              value = expr.release();
22          }
23          ;
24      ...
```

An *implication* expression is of the form $\beta_1(=> \beta_2(=> \cdots(=> \beta_k)))$ The parentheses show that operands of an implication associate to the right. We can express such right associative expressions by using *right recursion*, as in the following *Implication* rule:

```
1       ...
2       Implication(ParsingContext* ctx, var std::unique_ptr<Node> expr,
3           var Span s): Cm::Ast::Node*
4           ::=
5           (   Disjunction(ctx):left{ expr.reset(left); s = span; }
6               (   "=>"
7                   {
8                       if (!ctx->ParsingConcept()
9                       || ctx->ParsingTemplateId())
10                      pass = false;
11                  }
12                  Implication(ctx):right!
13                  {
14                      s.SetEnd(span.End());
15                      expr.reset(new ImplicationNode(s, expr.release(),
                            right));
16                  }
17              )?
18          )
19          {
20              value = expr.release();
21          }
22          ;
23      ...
```

A right recursive rule is of the form

$$p \rightarrow q\ (\ op\ p\ )?$$

where *op* is an operator that associates to the right. Like in *equivalence* expression, the implication expression of the form $\beta_1 => \beta_2$ is also accepted only in concept definitions. Sole *disjunction* expression β_1 is accepted always.

The *disjunction* rule rejects meaningless statements like $a||b = c;$, where $a||b$ is an *lvalue*. That is, when we are parsing the left part of an assignment statement, we set *parsingLvalue* flag is **true**, so in that case we reject expression of the form $a||b$.

```
1     ...
2     Disjunction(ParsingContext* ctx, var std::unique_ptr<Node> expr,
3         var Span s): Cm::Ast::Node*
4         ::=
5         (   Conjunction(ctx):left{ expr.reset(left); s = span; }
6             (   "||"
7                 {
8                     if (ctx->ParsingLvalue()
9                     || ctx->ParsingSimpleStatement()
10                        && !ctx->ParsingArguments())
11                        pass = false;
12                }
13                Conjunction(ctx):right!
14                {
15                    s.SetEnd(span.End());
16                    expr.reset(new DisjunctionNode(s, expr.release(),
                          right));
17                }
18            )*
19        )
20        {
21            value = expr.release();
22        }
23    ;
24    ...
```

Rules for other expressions are not shown, because there is nothing new in them. However, we show the syntax of *primary* expression. A *primary* expression consists one of

1. a parenthesized *expression*,
2. a *literal*,
3. a name of a basic type,
4. a **sizeof** expression,
5. a **cast** expression,
6. a **construct** expression,
7. a **new** expression,

8. a *template identifier*,
9. an *identifier*,
10. keyword **this**,
11. keyword **base** or a
12. **typename** expression.

```
1   Primary(ParsingContext* ctx): Cm::Ast::Node*
2       ::= ('(' Expression(ctx) ')'){ value = Expression; }
3       |   Literal{ value = Literal; }
4       |   BasicType{ value = BasicType; }
5       |   SizeOfExpr(ctx){ value = SizeOfExpr; }
6       |   CastExpr(ctx){ value = CastExpr; }
7       |   ConstructExpr(ctx){ value = ConstructExpr; }
8       |   NewExpr(ctx){ value = NewExpr; }
9       |   TemplateId(ctx){ value = TemplateId; }
10      |   Identifier{ value = Identifier; }
11      |   keyword("this"){ value = new ThisNode(span); }
12      |   keyword("base"){ value = new BaseNode(span); }
13      |   (keyword("typename") '(' Expression(ctx):subject ')')
14      {
15          value = new TypeNameNode(span, subject);
16      }
17      ;
```

3.6.5.5 Statements

The grammar for statements begins with rule link declarations:

```
1  grammar StatementGrammar
2  {
3      using stdlib.identifier;
4      using KeywordGrammar.Keyword;
5      using ExpressionGrammar.Expression;
6      using TypeExprGrammar.TypeExpr;
7      using IdentifierGrammar.Identifier;
8      using ExpressionGrammar.ArgumentList;
9      ...
```

Here is the definition of the *Statement* rule. There are brances for each kind of statement that Cmajor language contains.

```
1      ...
2      Statement(ParsingContext* ctx): Cm::Ast::StatementNode*
3          ::= LabeledStatement(ctx){ value = LabeledStatement; }
4          |   ControlStatement(ctx){ value = ControlStatement; }
5          |   TypedefStatement(ctx){ value = TypedefStatement; }
6          |   SimpleStatement(ctx){ value = SimpleStatement; }
7          |   AssignmentStatement(ctx){ value = AssignmentStatement; }
8          |   ConstructionStatement(ctx){ value = ConstructionStatement; }
9          |   DeleteStatement(ctx){ value = DeleteStatement; }
10         |   DestroyStatement(ctx){ value = DestroyStatement; }
11         |   ThrowStatement(ctx){ value = ThrowStatement; }
12         |   TryStatement(ctx){ value = TryStatement; }
13         |   AssertStatement(ctx){ value = AssertStatement; }
14         |   ConditionalCompilationStatement(ctx)
15             {
16                 value = ConditionalCompilationStatement;
17             }
18         ;
19     ...
```

The *SimpleStatement* rule consists of an optional expression. Thus it is the rule that matches also an empty statement consisting a sole semicolon.

```
1      ...
2      SimpleStatement(ParsingContext* ctx,
3          var std::unique_ptr<Node> expr): Cm::Ast::StatementNode*
4          ::= (empty{ ctx->PushParsingSimpleStatement(true); }
5              (Expression(ctx){ expr.reset(Expression); })? ';')
6          {
7              ctx->PopParsingSimpleStatement();
8              value = new SimpleStatementNode(span, expr.release());
9          }
10         /
11         {
12             ctx->PopParsingSimpleStatement();
13         }
14         ;
15     ...
```

The *ControlStatement* rule consists of cases for each kind of control statement.

```
1       ...
2       ControlStatement(ParsingContext* ctx): Cm::Ast::StatementNode*
3           ::= ReturnStatement(ctx){ value = ReturnStatement; }
4           |   ConditionalStatement(ctx){ value = ConditionalStatement; }
5           |   SwitchStatement(ctx){ value = SwitchStatement; }
6           |   WhileStatement(ctx){ value = WhileStatement; }
7           |   DoStatement(ctx){ value = DoStatement; }
8           |   RangeForStatement(ctx){ value = RangeForStatement; }
9           |   ForStatement(ctx){ value = ForStatement; }
10          |   CompoundStatement(ctx){ value = CompoundStatement; }
11          |   BreakStatement(ctx){ value = BreakStatement; }
12          |   ContinueStatement(ctx){ value = ContinueStatement; }
13          |   GotoCaseStatement(ctx){ value = GotoCaseStatement; }
14          |   GotoDefaultStatement(ctx){ value = GotoDefaultStatement; }
15          |   GotoStatement(ctx){ value = GotoStatement; }
16          ;
17      ...
```

We are showing just the definition of the return statement and while statement rules.

A return statement consists of keyword **return** followed by an optional expression and a semicolon. The *ReturnStatement* rule constructs an abstract syntax tree node called *ReturnStatementNode*, that takes the input position and synhesized attribute of the *Expression* rule as arguments, and assigns it to the synthesized attribute of the rule. The exclamation mark after the semicolon disables backtracking. If the semicolon is missing in input, an *ExpectationFailure* exception containing exact input position is thrown.

```
1       ...
2       ReturnStatement(ParsingContext* ctx): Cm::Ast::StatementNode*
3           ::= (keyword("return") Expression(ctx)? ';'!)
4           {
5               value = new ReturnStatementNode(span, Expression);
6           }
7           ;
8       ...
```

A while statement consists of keyword **while**, a Boolean expression and a statement. The exclamation marks after the parentheses, and the calls of the expression rule and statement rule disable backtracking and force matching those constructs. The *WhileStatement* rule constructs an abstract syntax tree node called *WhileStatementNode* that takes the synthesized attributes of the *Expression* and *Statement* rules as arguments, and assigns it to the synthesized attribute of the rule.

```
1       ...
2       WhileStatement(ParsingContext* ctx): Cm::Ast::StatementNode*
3           ::= (keyword("while") '('! Expression(ctx)! ')'! Statement(ctx)!)
4           {
5               value = new WhileStatementNode(span, Expression, Statement);
6           }
7           ;
8       ...
```

3.6.6 Abstract Syntax Tree Class Hierarchy

There are three abstract node classes in the abstract syntax tree node class hierarchy: *Node*, *UnaryNode* and *BinaryNode*.

The *Node* class is the root of the abstract syntax tree node class hierarchy. The *UnaryNode* class is an abstract syntax tree node that has one child node. The *BinaryNode* class is an abstract syntax tree node that has two child nodes.

```
Node
    UnaryNode
    BinaryNode
```

3.6.6.1 Node Classes for Basic Types

There is a node class for each basic type:

```
Node
    BoolNode
    SByteNode
    ByteNode
    ShortNode
    UShortNode
    IntNode
    UIntNode
    LongNode
    ULongNode
    FloatNode
    DoubleNode
    CharNode
    WCharNode
    UCharNode
    VoidNode
```

3.6.6.2 Literal Node Classes

There is a node class for each kind of literal:

```
Node
    BooleanLiteralNode
    SByteLiteralNode
    ByteLiteralNode
    ShortLiteralNode
    UShortLiteralNode
    IntLiteralNode
    UIntLitralNode
    LongLitralNode
    ULongLiteralNode
    FloatLiteralNode
```

```
    DoubleLiterallNode
    CharLiteralNode
    StringLiteralNode
    WStringLiteralNode
    UStringLiteralNode
    NullLiteralNode
```

3.6.6.3 Expression Node Classes

There is a node class for each kind of Cmajor expression:

```
Node
    CastNode
    IsNode
    AsNode
    NewNode
    ConstructNode
    ThisNode
    BaseNode
    UnaryNode
        InvokeNode
        IndexNode
        DotNode
        ArrowNode
        PostfixIncNode
        PostfixDecNode
        DerefNode
        AddOfNode
        NotNode
        UnaryPlusNode
        UnaryMinusNode
        ComplementNode
        PrefixIncNode
        PrefixDecNode
        SizeOfNode
        TypeNameNode
    BinaryNode
        EquivalenceNode
        ImplicationNode
        DisjunctionNode
        ConjunctionNode
        BitOrNode
        BitXorNode
        BitAndNode
        EqualNode
        NotEqualNode
        LessNode
```

```
        GreaterNode
        LessOrEqualNode
        GreaterOrEqualNode
        ShiftLeftNode
        ShirtRightNode
        AddNode
        SubNode
        MulNode
        DivNode
        RemNode
```

3.6.6.4 Statement Node Classes

There is a node class for each kind of Cmajor statement:

```
Node
    LabelNode
    CatchNode
    CondCompSymbolNode
    CondCompilationPartNode
    CondCompExprNode
        CondCompNotNode
        CondCompPrimaryNode
        CondCompBinExprNode
            CondCompDisjunctionNode
            CondCompConjunctionNode
    StatementNode
        SimpleStatementNode
        ReturnStatementNode
        ConditionalStatementNode
        SwitchStatementNode
        CaseStatementNode
        DefaultStatementNode
        GotoCaseStatementNode
        GotoDefaultStatementNode
        WhileStatementNode
        DoStatementNode
        ForStatementNode
        RangeForStatementNode
        CompoundStatementNode
        BreakStatementNode
        ContinueStatementNode
        GotoStatementNode
        TypedefStatementNode
        AssignmentStatementNode
        ConstructionStatementNode
        DeleteStatementNode
```

```
        DestroyStatementNode
        ThrowStatementNode
        TryStatementNode
        ExitTryStatementNode
        BeginCatchStatementNode
        AssertStatementNode
        CondCompStatementNode
```

3.6.6.5 Concept Node Classes

Node classes relating to concepts:

```
Node
    AxiomStatementNode
    AxiomNode
    ConceptIdNode
    ConceptNode
        SameConceptNode
        DerivedConceptNode
        ConvertibleConceptNode
        ExplicitlyConvertibleConceptNode
        CommonConceptNode
        NonReferenceTypeConceptNode
    ConstraintNode
        WhereConstraintNode
        IsConstraintNode
        MultiParamConstraintNode
        TypeNameConstraintNode
        IntrinsicConstraintNode
            SameConstraintNode
            DerivedConstaraintNode
            ConvertibleConstraintNode
            ExplicitlyConvertibleConstraintNode
            CommonConstraintNode
            NonReferenceTypeConstraintNode
        SignatureConstraintNode
            ConstructorConstraintNode
            DestructorConstraintNode
            MemberFunctionConstraintNode
            FunctionConstraintNode
        BinaryConstraintNode
            DisjunctiveConstraintNode
            ConjunctiveConstraintNode
```

3.6.6.6 Class and Function Node Classes

Node classes relating to classes and functions:

```
Node
    MemberVariableNode
    FunctionGroupIdNode
    FunctionNode
        StaticConstructorNode
        ConstructorNode
        DestructorNode
        MemberFunctionNode
            ConversionFunctionNode
    ClassNode
    InitializerNode
        MemberInitializerNode
        BaseInitializerNode
        ThisInitializerNode
```

3.6.6.7 Other Node Classes

Other kinds of node classes:

```
Node
    CompileUnitNode
    ConstantNode
    DelegateNode
    ClassDeletateNode
    DerivedTypeExprNode
    EnumConstantNode
    EnumTypeNode
    IdentifierNode
    InterfaceNode
    NamespaceNode
    AliasNode
    NamespaceImportNode
    ParameterNode
    TemplateParameterNode
    TemplateIdNode
    TypedefNode
```

3.6.7 Example

The following example shows the result of parsing a function and constructing an abstract syntax tree for it.

Example 3.6.3. The following Cmajor function is used as example input to the parser:

```
public nothrow int StrLen(const char* s)
{
    int len = 0;
    if (s != null)
    {
        while (*s != '\0')
        {
            ++len;
            ++s;
        }
    }
    return len;
}
```

The following listing shows the resulting abstract syntax tree for parsing the *StrLen* function:

```
CompileUnitNode
    NamespaceNode()
        FunctionNode
            FunctionGroupIdNode(StrLen)
            ParameterNodeList
                ParameterNode
                    DerivedTypeExprNode
                        DerivationList
                            Derivation.const
                            Derivation.pointer
                        CharNode
                    IdentifierNode(s)
            CompoundStatementNode
                ConstructionStatementNode
                    IntNode
                    IdentifierNode(len)
                    SByteLiteralNode(0)
                ConditionalStatementNode
                    NotEqualNode
                        IdentifierNode(s)
                        NullLiteralNode
                    CompoundStatementNode
                        WhileStatementNode
                            NotEqualNode
                                DerefNode
                                    IdentifierNode(s)
```

```
                        CharLiteralNode('\0')
                    CompoundStatementNode
                        SimpleStatementNode
                            PrefixIncNode
                                IdentifierNode(len)
                        SimpleStatementNode
                            PrefixIncNode
                                IdentifierNode(s)
            ReturnStatementNode
                IdentifierNode(len)
```

The parser constructs an abstract syntax tree node called *FunctionNode* for the function. The FunctionNode contains:

1. the name of the function group that the function belongs to: *FunctionGroupIdNode*(StrLen).
2. nodes for each parameter that the function takes. Each *ParameterNode* consists of nodes for the type and the name of the parameter.
3. node for the body of the function: *CompoundStatementNode*.

The body consists of an construction statement, an **if** statement and a return statement. The **if** statement consists of a **while** statement that has two simple statements in it. Each simple statement contains a prefix increment expression.

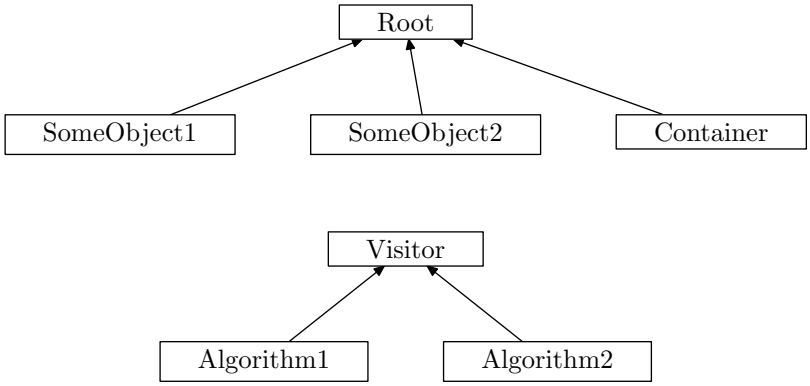
3.7 Iterating Through the Abstract Syntax Trees using Visitor Design Pattern

Many of the following phases of compilation iterate through the abstract syntax trees generated by the parser. Technically the iteration is done using the *visitor* design pattern.

The visitor design pattern enables creation of several algorithms that operate on a object hierarchy without touching the object hierarchy. In visitor pattern, each object that is part of the object hierarchy implements a virtual *Accept* member function that takes a parameter of a class derived from common *Visitor* class. Accept calls *Visit* member function of a visitor by passing itself as a parameter to the Visit member function.

```
1  class Root
2  {
3  public:
4      virtual void Accept(Visitor& visitor) = 0;
5  };
6
7  class SomeObject1 : public Root
8  {
9  public:
10     void Accept(Visitor& visitor) override
11     {
12         visitor.Visit(*this);
13     }
14 };
```

Figure 3.13: Visitor




```
15
16 class SomeObject2 : public Root
17 {
18 public:
19     void Accept(Visitor& visitor) override
20     {
21         visitor.Visit(*this);
22     }
23 };
24
25 class Container : public Root
26 {
27 public:
28     void Accept(Visitor& visitor) override
29     {
30         o1->Accept(visitor);
31         o2->Accept(visitor);
32         visitor.Visit(*this);
33     }
34 private:
35     SomeObject1* o1;
36     SomeObject2* o2;
37 };
38
39 class Visitor
40 {
41 public:
42     virtual void Visit(SomeObject1& someObject1) {}
43     virtual void Visit(SomeObject2& someObject2) {}
44     virtual void Visit(Container& container) {}
45 };
```

```
1  class Algorithm1 : public Visitor
2  {
3  public:
4      void Visit(SomeObject1& someObject1) override
5      {
6          // algorithm 1 for SomeObject1
7      }
8      void Visit(SomeObject2& someObject2) override
9      {
10         // algorithm 1 for SomeObject2
11     }
12     void Visit(Container& container) override
13     {
14         // algorithm 1 for Container
15     }
16 };
17
18 class Algorithm2 : public Visitor
19 {
20 public:
21     void Visit(SomeObject1& someObject1) override
22     {
23         // algorithm 2 for SomeObject1
24     }
25     void Visit(SomeObject2& someObject2) override
26     {
27         // algorithm 2 for SomeObject2
28     }
29     void Visit(Container& container) override
30     {
31         // algorithm 2 for Container
32     }
33 };
34
35 void DoAlgorithm1(Container& c)
36 {
37     Algorithm1 algorithm1;
38     c.Accept(algorithm1);
39 }
40
41 void DoAlgorithm2(Container& c)
42 {
43     Algorithm2 algorithm2;
44     c.Accept(algorithm2);
45 }
```

3.7.1 Visitor Pattern Applied in Cmajor

In Cmajor the visitor pattern is extended by providing two visiting points for containers. When starting to visit a container, visitor's `BeginVisit(Container&)` member function is called. Then the contained elements are visited by calling their `Accept` member functions. Finally, when ending to visit a container, visitor's `EndVisit(Container&)` member function

is called.

Example 3.7.1. Visiting a Namespace in Cmajor.

```
class NamespaceNode
{
public:
    virtual void Accept(Visitor& visitor)
    {
        visitor.BeginVisit(*this);
        for (Node* node : members)
        {
            node->Accept(visitor);
        }
        visitor.EndVisit(*this);
    }
private:
    std::vector<Node*> members;
};
```

Chapter 4

Symbol Table

The next phase of compilation after parsing is constructing a symbol table.

4.1 Symbol Table Structure

A symbol table consists of a tree of symbols. There are many kinds of symbols. Container symbols like class and namespace symbols form the interior nodes of the symbol tree. Simple kind of symbols like constant and parameter symbols form the leaf nodes of the symbol tree.

4.1.1 Symbol Class Hierarchy

The following listing shows the most important kind of symbols:

```
Symbol
    FunctionGroupSymbol
    ConceptGroupSymbol
    ConstantSymbol
    EnumConstantSymbol
    TypedefSymbol
    VariableSymbol
        LocalVariableSymbol
        MemberVariableSymbol
        ParameterSymbol
    ContainerSymbol
        FunctionSymbol
        NamespaceSymbol
        ConceptSymbol
        TypeSymbol
            BasicTypeSymbol
                ...
            ClassTypeSymbol
                TemplateTypeSymbol
            DerivedTypeSymbol
            EnumTypeSymbol
            InterfaceTypeSymbol
```

4.1.2 Properties of Symbols

We inspect properties of symbols that make possible symbol algorithms.

4.1.2.1 Properties Common To All Symbols

The most important attribute common to each kind of symbol is its name. Another property common to all symbols is a pointer to the symbol's parent symbol in the symbol table. The global namespace symbol is the root of the symbol tree. The name of the global namespace symbol is empty and its parent property is null. Other symbols have a nonempty name and a nonnull parent property.

With the name and parent properties the *full name* of a symbol can be computed. The algorithm returns a string that consists of nonempty names of symbols along a path from the global namespace symbol to the symbol separated by dot characters. Fox example, the full name of the global namespace symbol is an empty string, and the full name of a class symbol whose name is `gamma` that is contained by namespace symbol whose name is `beta` that is contained by a namespace symbol whose name is `alpha` that is contained by the global namespace symbol is `alpha.beta.gamma`.

Algorithm 4.1.1. Computing the Full Name of a Symbol.

1. If the symbol's parent property is not null let p be the full name of symbol's parent. Otherwise let p be empty string.

2. If p is empty string, the full name of the symbol is the name of the symbol. Otherwise the full name of the symbol is p concatenated with "." and the name of the symbol.

With the parent property also an associated namespace symbol for a symbol can be computed as follows:

Algorithm 4.1.2. Computing an Associated Namespace Symbol for a Symbol.

1. Let s be the symbol for which to compute the associated namespace symbol.

2. If s is a namespace symbol, return s.

3. Otherwise, if the parent symbol of s is not null, compute the associated namespace symbol for the parent symbol of s and return it.

4. Otherwise, throw an exception.

4.1.2.2 Properties of Container Symbols

Each container symbol, say S, like a namespace or a class symbol, has a *container scope*, say C, that keeps a mapping from names of contained symbols to contained symbols themselves. A container scope C also has pointers to its *base scope* and its *parent scope*. If S is a class type symbol, the base scope of C is the container scope of the base class symbol of S. The parent scope of C is the container scope of the parent symbol of S. A container scope also contains a pointer to its owning container symbol.

4.1.3 Symbol Name Lookup

Symbol name lookup searches a symbol from a number of container scopes using a possibly qualified name, a scope kind, and kinds of a symbols to search. A scope kind is a combination of following values: **this**, **base** and **parent**. The kind of symbol to search can be one of many values. For example, lookup: **all** symbols, only **type** symbols, only **namespace** symbols, only **variable** and **parameter** symbols, etc.

Let $c_1, \ldots, c_n$ be the components of a qualified name to search. The components are separated by dots. For example, if the name to search is `alpha`, then $n = 1$ and $c_1 =$ `alpha`. Another example: if the name to search is `alpha.beta`, then $n = 2$, $c_1 =$ `alpha` and $c_2 =$ `beta`.

4.1.3.1 Unqualified Name Lookup

If $n = 1$, we have a simple name and the symbol name lookup performs an *unqualified name lookup*:

Algorithm 4.1.3. Unqualified Name Lookup. The algorithm returns a symbol if the search is successful, or null otherwise.

1. Let s be the name to search, t be the container scope from which the search begins, p be the set of scope kinds to search, and k be the kind of symbol to search.
2. If s is found from the mapping of t, let m be the mapped symbol. If the symbol kind of m is equal to k, return symbol m.
3. If p contains the **base** scope and the base scope of t is not null, perform unqualified name lookup from the base scope of t. If that search is successful, return the symbol found.
4. If p contains the **parent** scope and the parent scope of t is not null, perform unqualified name lookup from the parent scope of t. If that search is successful, return the symbol found.
5. Otherwise, return null.

4.1.3.2 Qualified Name Lookup

If $n > 1$, we have a qualified name of at least two components and the symbol name lookup performs a *qualified name lookup*:

Algorithm 4.1.4. Qualified Name Lookup. The algorithm returns a symbol if the search is successful, or null otherwise.

1. Let $c_1, \ldots, c_n$ be the components of a qualified name to search ($n > 1$), t and u be the container scope from which the search begins, p be the set of scope kinds to search, and k be the kind of symbol to search, flag a be **true**, symbol s be null.
2. For $i = 1, \ldots, n$:
   (a) If t is not null, perform unqualified name lookup (algorithm 4.1.3) for name c_i, using scope t and scope kind **this**. If $i < n$, set the kind of symbol to search to

only **container** symbols, otherwise, if $i = n$, set the kind of symbol to search to k. If the search was successful, let s be the returned symbol, and let t be the container scope of s. Otherwise let t be null and let a be **false**.

3. If s is null or a is **false**, and if **parent** scope is in p and the parent scope of u is not null, perform qualified name lookup (this algorithm) for the parent scope of u. If the search was successful, return the symbol found. Otherwise return null.

4. Otherwise return symbol s.

4.1.4 Opening and Closing Container Symbols

A symbol table also keeps track of *currently open container symbol* and a *stack of open container symbols*. Initially the currently open container symbol is the global namespace symbol (the root of the symbol tree), and the stack of open container symbols is empty.

A nonnamespace container symbol is opened by pushing the currently open container symbol to the stack of open container symbols, and then setting the container symbol as the currently open container symbol. Any container symbol is closed by popping a container symbol from the stack of open container symbols and setting it as the currently open container symbol.

4.1.4.1 Opening a Namespace

A namespace is opened using the following algorithm:

Algorithm 4.1.5. Opening a Namespace. The algorithm sets the currently open container symbol of a symbol table.

1. Let n be a possibly qualified namespace name to open. Let t be a symbol table to which to open the namespace.

2. If n is an empty string, push the currently open container symbol to the stack of open container symbols of t, and then set the global namespace symbol as the currently open container symbol of t.

3. Otherwise lookup n (see section 4.1.3) from the container scope of the currently open container symbol using scope kind **this** and setting the kind of symbols to search as **namespace** symbols. If the search was successful, let s be the symbol found, otherwise let s be null.

4. If s is a namespace symbol, push the currently open container symbol to the stack of open container symbols of t, and then set s as the currently open container symbol of t. Otherwise if s is not a namespace symbol, throw an exception.

5. Otherwise s is null, so use algorithm 4.1.6 to create a namespace to the container scope of the currently open container symbol of t, and open it by pushing the currently open container symbol to the stack of open container symbols of t, and then set the created namespace symbol as the currently open container symbol of t.

4.1.4.2 Creating a Namespace

A namespace is created using the following algorithm:

Algorithm 4.1.6. Creating a Namespace. The algorithm returns created namespace symbol.

1. Let m be a possibly qualified namespace name to create and let t be the container scope to which the namespace symbol is to be created. Let $c_1, \ldots, c_n$ be the n components of m separated by dots. Let p be the namespace symbol associated with the owner symbol of the container scope t. It can be computed using algorithm 4.1.2.
2. For $i = 1, \ldots, n$:
   (a) Lookup name c_i (see section 4.1.3) from container scope t using scope kind **this** and setting the kind of symbols to search as **namespace** symbols. If the search was successful let s be the symbol found. Otherwise let s be null.
   (b) If s is not null and s is a namespace symbol, let t be the container scope of s and let p be the namespace symbol associated with the owner symbol of the container scope t (algorithm 4.1.2). Otherwise if s is not null and s is not a namespace symbol, throw an exception.
   (c) Otherwise s is null, so create a new namespace symbol ns with name c_i. Let t be the container scope of ns. Let the parent scope of t be the container scope of p. Add symbol ns as the child symbol of p. Finally let p be ns.
3. Return p.

4.1.5 Adding Symbols to Containers

A symbol is added as a child of a container symbol using the following algorithm:

Algorithm 4.1.7. Adding a Symbol to a Container.

1. Let s be the symbol to add to a container symbol c.
2. If the name of s is not empty and s is not a function symbol and s is not a concept symblol and s is not a declaration block symbol and s is not a namespace type symbol, install the symbol to the container scope of c using following steps:
   (a) If the name of s is found from symbol name mappings of the container scope of c, throw an exception, because the name of a symbol must be unique in its immediate container.
   (b) Add a mapping from name of s to s to the $name \rightarrow symbol$ mapping of the container scope of c.
   (c) If symbol is a container symbol, set the parent scope of the container scope of s to the container scope of c.
3. If s if a function symbol, open a function group using the group name of s and add s to the opened function group using algorithm 4.1.8.
4. Otherwise, if s is a concept symbol, open a concept group using the group name of s and add s to the opened concept group using algorithm 4.1.9.
5. Othwerwise, add s as a child symbol of c and set the parent property of s to c.

4.1.5.1 Function Groups

Function symbols are not added directly to containers, but there is an extra layer called a *function group* in between the container symbol and the function symbol. To describe function groups we need two definitions:

Definition 4.1.1. The *group name* of a nonmember function is the name of the function without its parameters. The group name of a constructor is "@constructor" and the group name a destructor is "@destructor". The group name of other member function is the name of the member function without its parameters. For example, the group name of function

```
void foo(int x, double y)
```

is `foo` and the group name of member function

```
void C.operator=(const C& x)
```

is `operator=`.

Definition 4.1.2. The *arity* of a function is the number of its parameters. For example, the arity of function

```
void foo(int x, double y)
```

is 2.

A function group collects functions that have equal group name under a name. A function group has a mapping from arities of functions to lists of function symbols.

Example 4.1.1. For example, if we have three functions:

```
void foo(int a);
void foo(double b);
void foo(int a, double b);
```

they all belong to a function group named *foo*. The function group *foo* contains a mapping from arity 1 to a list containing two functions: `void foo(int a)` and `void foo(double b)`. It also contains a mapping from arity 2 to a list containing one function: `void foo(int a, double b)`.

Opening a function group and adding a function to it is performed using the following algorithm:

Algorithm 4.1.8. Opening a Function Group, and Adding a Function to it.

1. Let s be a function symbol to add to a function group under container c.
2. Lookup the group name of s from the container scope of c using scope kind **this** (algorithm 4.1.3). If the search was successful, let g be symbol found. Otherwise let g be null.
3. If g is null, create a new function group symbol using group name of s, and add it to c using algorithm 4.1.7. Let g be the created function group.
4. Otherwise, if g is not a function group symbol, throw an exception, because name of a function group conflicts with name of another symbol.
5. Let a be the arity of s. Add the s to a list of functions of arity a in the $arity \rightarrow list$ mappings of g.
6. Add s as a child symbol of g.

4.1.5.2 Concept Groups

What is said about functions and function groups applies analogically to concepts and concept groups. Concept group acts as a layer between a container and a concept symbol. Also analogically to a group name and arity of a function, we can define the group name and arity of a concept as follows:

Definition 4.1.3. The *group name* of a concept is the name of a concept without its type parameters. For example, the group name of concept

```
EqualityComparable<T, U>
```

is `EqualityComparable`.

Definition 4.1.4. The *arity* of a concept is the number of its type parameters. For example, the arity of concept

```
EqualityComparable<T, U>
```

is 2.

A concept group collects concepts that have equal group name under a name. A concept group has a mapping from arities of concepts to concept symbols.

Example 4.1.2. For example, if we have these two concepts:

```
EqualityComparable<T>
EqualityComparable<T, U>
```

they both belong to a concept group named *EqualityComparable*. The concept group *EqualityComparable* contains a mapping from arity 1 to a concept symbol `EqualityComparable<T>` and from arity 2 to a concept symbol `EqualityComparable<T, U>`.

Opening a concept group and adding a concept to it is performed using the following algorithm:

Algorithm 4.1.9. Opening a Concept Group, and Adding a Concept to it.

1. Let s be a concept symbol to add to a concept group under container c.
2. Lookup the group name of s from the container scope of c using scope kind **this** (algorithm 4.1.3). If the search was successful, let g be symbol found. Otherwise let g be null.
3. If g is null, create a new concept group symbol using group name of s, and add it to c using algorithm 4.1.7. Let g be the created concept group.
4. Otherwise, if g is not a concept group symbol, throw an exception, because name of a concept group conflicts with name of another symbol.
5. Let a be the arity of s. Set s as a concept for arity a in the $arity \rightarrow concept$ mappings of g.
6. Add s as a child symbol of g.

4.2 Construction of the Global Symbol Table

The global symbol table is built in three stages:

1. First basic type symbols like *BoolTypeSymbol* and *IntTypeSymbol*, and functions that operate on them, are inserted to the global namespace of the global symbol table.
2. Then the symbol tables of the referenced libraries are read and imported to the global symbol table.
3. Finally the abstract syntax trees of the project being compiled are iterated and symbols that correspond abstract syntax tree nodes are created and inserted to the global symbol table.

4.2.1 Insertion of Basic Types and Their Operations

The first stage in constructing the global symbol table is inserting the basic types and their operations to the global symbol table. The following listing shows the basic type symbols that are inserted to the global namespace of the global symbol table:

```
BasicTypeSymbol
    BoolTypeSymbol
    CharTypeSymbol
    WCharTypeSymbol
    UCharTypeSymbol
    VoidTypeSymbol
    SByteTypeSymbol
    ByteTypeSymbol
    ShortTypeSymbol
    UShortTypeSymbol
    IntTypeSymbol
    UIntTypeSymbol
    LongTypeSymbol
    ULongTypeSymbol
    FloatTypeSymbol
    DoubleTypeSymbol
    NullPtrTypeSymbol
```

The following listing shows operations for basic types that are inserted to the global namespace of the global symbol table: [1]

```
Symbol
    ContainerSymbol
        FunctionSymbol
            BasicTypeOp
                DefaultCtor[@constructor]
                CopyCtor[@constructor]
```

[1] The group name of the function symbol is shown in brackets after the operation.

```
CopyAssignment[operator=]
MoveCtor[@constructor]
MoveAssignment[operator=]
OpEqual[operator==]
OpLess[operator<]
BinOp
    OpAdd[operator+]
    OpSub[operator-]
    OpMul[operator*]
    OpDiv[operator/]
    OpRem[operator%]
    OpShl[operator<<]
    OpShr[operator>>]
    OpBitAnd[operator&]
    OpBitOr[operator|]
    OpBitXor[operator^]
OpNot[operator!]
OpUnaryPlus[operator+]
OpUnaryMinus[operator-]
OpComplement[operator~]
OpIncrement[operator++]
OpDecrement[operator--]
ConvertingCtor[@constructor]
```

4.2.1.1 Operations for bool

Operations for *BoolTypeSymbol* are: *DefaultCtor*, *CopyCtor*, *CopyAssignment*, *MoveCtor*, *MoveAssignment*, *OpEqual*, *OpLess*, *OpNot*.

4.2.1.2 Operations for Integer Types

Operations for integer types (*SByteTypeSymbol*, *ByteTypeSymbol*, *ShortTypeSymbol*, *UShortTypeSymbol*, *IntTypeSymbol*, *UIntTypeSymbol*, *LongTypeSymbol*, *ULongTypeSymbol*) are: *DefaultCtor*, *CopyCtor*, *CopyAssignment*, *MoveCtor*, *MoveAssignment*, *OpEqual*, *OpLess*, *OpAdd*, *OpSub*, *OpMul*, *OpDiv*, *OpRem*, *OpShl*, *OpShr*, *OpBitAnd*, *OpBitOr*, *OpBitXor*, *OpUnaryPlus*, *OpUnaryMinus*, *OpComplement*, *OpIncrement*, *OpDecrement*.

4.2.1.3 Operations for Floating Point Types

Operations for floating point types (*FloatTypeSymbol* and *DoubleTypeSymbol*) are: *DefaultCtor*, *CopyCtor*, *CopyAssignment*, *MoveCtor*, *MoveAssignment*, *OpEqual*, *OpLess*, *OpAdd*, *OpSub*, *OpMul*, *OpDiv*, *OpUnaryPlus*, *OpUnaryMinus*.

4.2.1.4 Operations for Character Types

Operations for character types (*CharTypeSymbol*, *WCharTypeSymbol* and *UCharTypeSymbol*) are: *DefaultCtor*, *CopyCtor*, *CopyAssignment*, *MoveCtor*, *MoveAssignment*, *OpEqual*, *OpLess*.

4.2.1.5 Standard Conversions

The following table shows standard conversion operations (*ConvertingCtor*) that are inserted to the global namespace of the global symbol table.

Abbreviations are:
I - implicit conversion,
E - explicit conversion,
C - conversion,
P - promotion

Target Type	Source Type	Explicit/Implicit	Rank	Distance
sbyte	**byte**	E	C	
sbyte	**short**	E	C	
sbyte	**ushort**	E	C	
sbyte	**int**	E	C	
sbyte	**uint**	E	C	
sbyte	**long**	E	C	
sbyte	**ulong**	E	C	
sbyte	**float**	E	C	
sbyte	**double**	E	C	
sbyte	**char**	E	C	
sbyte	**wchar**	E	C	
sbyte	**uchar**	E	C	
sbyte	**bool**	E	C	
byte	**sbyte**	E	C	
byte	**short**	E	C	
byte	**ushort**	E	C	
byte	**int**	E	C	
byte	**uint**	E	C	
byte	**long**	E	C	
byte	**ulong**	E	C	
byte	**float**	E	C	
byte	**double**	E	C	
byte	**char**	E	C	
byte	**wchar**	E	C	
byte	**uchar**	E	C	
byte	**bool**	E	C	
short	**sbyte**	I	P	1
short	**byte**	I	P	2
short	**ushort**	E	C	
short	**int**	E	C	
short	**uint**	E	C	
short	**long**	E	C	

short	**ulong**	E	C	
short	**float**	E	C	
short	**double**	E	C	
short	**char**	E	C	
short	**wchar**	E	C	
short	**uchar**	E	C	
short	**bool**	E	C	
ushort	**sbyte**	E	C	
ushort	**byte**	I	P	1
ushort	**short**	E	C	
ushort	**int**	E	C	
ushort	**uint**	E	C	
ushort	**long**	E	C	
ushort	**ulong**	E	C	
ushort	**float**	E	C	
ushort	**double**	E	C	
ushort	**char**	E	C	
ushort	**wchar**	E	C	
ushort	**uchar**	E	C	
ushort	**bool**	E	C	
int	**sbyte**	I	P	3
int	**byte**	I	P	4
int	**short**	I	P	1
int	**ushort**	I	P	2
int	**uint**	E	C	
int	**long**	E	C	
int	**ulong**	E	C	
int	**float**	E	C	
int	**double**	E	C	
int	**char**	E	C	
int	**wchar**	E	C	
int	**uchar**	E	C	
int	**bool**	E	C	
uint	**sbyte**	E	C	
uint	**byte**	I	P	2
uint	**short**	E	C	
uint	**ushort**	I	P	1
uint	**int**	E	C	
uint	**long**	E	C	
uint	**ulong**	E	C	
uint	**float**	E	C	
uint	**double**	E	C	

uint	**char**	E	C	
uint	**wchar**	E	C	
uint	**uchar**	E	C	
uint	**bool**	E	C	
long	**sbyte**	I	P	5
long	**byte**	I	P	6
long	**short**	I	P	3
long	**ushort**	I	P	4
long	**int**	I	P	1
long	**uint**	I	P	2
long	**ulong**	E	C	
long	**float**	E	C	
long	**double**	E	C	
long	**char**	E	C	
long	**wchar**	E	C	
long	**uchar**	E	C	
long	**bool**	E	C	
ulong	**sbyte**	E	C	
ulong	**byte**	I	P	3
ulong	**short**	E	C	
ulong	**ushort**	I	P	2
ulong	**int**	E	C	
ulong	**uint**	I	P	1
ulong	**long**	E	C	
ulong	**float**	E	C	
ulong	**double**	E	C	
ulong	**char**	E	C	
ulong	**wchar**	E	C	
ulong	**uchar**	E	C	
ulong	**bool**	E	C	
float	**sbyte**	I	C	5
float	**byte**	I	C	6
float	**short**	I	C	3
float	**ushort**	I	C	4
float	**int**	I	C	1
float	**uint**	I	C	2
float	**long**	E	C	
float	**ulong**	E	C	
float	**double**	E	C	
float	**char**	E	C	
float	**wchar**	E	C	
float	**uchar**	E	C	

float	**bool**	E	C	
double	**sbyte**	I	C	8
double	**byte**	I	C	9
double	**short**	I	C	6
double	**ushort**	I	C	7
double	**int**	I	C	4
double	**uint**	I	C	5
double	**long**	I	C	2
double	**ulong**	I	C	3
double	**float**	I	P	1
double	**char**	E	C	
double	**wchar**	E	C	
double	**uchar**	E	C	
double	**bool**	E	C	
char	**sbyte**	E	C	
char	**byte**	E	C	
char	**short**	E	C	
char	**ushort**	E	C	
char	**int**	E	C	
char	**uint**	E	C	
char	**long**	E	C	
char	**ulong**	E	C	
char	**float**	E	C	
char	**double**	E	C	
char	**wchar**	E	C	
char	**uchar**	E	C	
char	**bool**	E	C	
wchar	**sbyte**	E	C	
wchar	**byte**	E	C	
wchar	**short**	E	C	
wchar	**ushort**	E	C	
wchar	**int**	E	C	
wchar	**uint**	E	C	
wchar	**long**	E	C	
wchar	**ulong**	E	C	
wchar	**float**	E	C	
wchar	**double**	E	C	
wchar	**char**	I	P	1
wchar	**uchar**	E	C	
wchar	**bool**	E	C	
uchar	**sbyte**	E	C	
uchar	**byte**	E	C	

uchar	**short**	E	C	
uchar	**ushort**	E	C	
uchar	**int**	E	C	
uchar	**uint**	E	C	
uchar	**long**	E	C	
uchar	**ulong**	E	C	
uchar	**float**	E	C	
uchar	**double**	E	C	
uchar	**char**	I	P	2
uchar	**wchar**	I	P	1
uchar	**bool**	E	C	
bool	**sbyte**	E	C	
bool	**byte**	E	C	
bool	**short**	E	C	
bool	**ushort**	E	C	
bool	**int**	E	C	
bool	**uint**	E	C	
bool	**long**	E	C	
bool	**ulong**	E	C	
bool	**float**	E	C	
bool	**double**	E	C	
bool	**char**	E	C	
bool	**wchar**	E	C	
bool	**uchar**	E	C	

4.2.2 Importing Symbol Tables of Referenced Libraries

The second stage in constructing the global symbol table is reading the symbol table of referenced libraries and importing the symbols from them into the global symbol table. For each library L that the project being compiled references,

- the symbol table u of L is read from the library file of L and then
- the symbols from the global namespace of symbol table u are imported into the global symbol table using algorithm 4.2.1.

Algorithm 4.2.1. Importing Symbols from a Namespace into a Symbol Table.

1. Let t be the symbol table to which the the symbols are to be imported. Let n be the source namespace, i.e. the namespace from which the symbols are to be imported.
2. Open a namespace of name of n to the symbol table t using algorithm 4.1.5.
3. For each child symbol c of n:
   (a) If c is a namespace symbol, import c to t by calling this algorithm recursively.
   (b) Otherwise, add c to the currently open container symbol of t using algorithm 4.1.7.
4. Close the currently open namespace of t.

4.2.3 Creating Symbols for the Project Being Compiled

The third stage in constructing the global symbol table is creating the symbols for the project being compiled and inserting them to the global symbol table. For that we need four auxiliary algorithms:

Algorithm 4.2.2. Opening a Function Scope.

1. Let n be the function node for which to open the function scope.
2. Create a function symbol with the name defined in n and set its group name to the group name in n.
3. Open the function symbol as described in section 4.1.4.

Algorithm 4.2.3. Closing a Function Scope.

1. Let f be the function symbol to close.
2. Close f as described in section 4.1.4.
3. Add f to the currentlty open container symbol using algorithm 4.1.7.

Algorithm 4.2.4. Opening a Declaration Scope.

1. Let s be the statement node for which to open the declaration scope.
2. Create a declaration block symbol and open it as described in section 4.1.4.

Algorithm 4.2.5. Closing a Declaration Scope.

Let d be the declaration block symbol to close.

Close d as described in section 4.1.4.

Creating and adding symbols to the global symbol table is done using the following algorithm:

Algorithm 4.2.6. Creating and Adding Symbols to the Global Symbol Table. The algorithm is implemented by an abstract syntax tree visitor called declaration visitor. The declaration visitor visits abstract syntax tree nodes by overriding the following visiting points:

- `BeginVisit(NamespaceNode& namespaceNode)`: Open a namespace for possibly qualified namespace name defined in the namespace node using algorithm 4.1.5.
- `EndVisit(NamespaceNode& namespaceNode)`: Close the currently open namespace of the global symbol table as described in section 4.1.4.
- `BeginVisit(ClassNode& classNode)`: Create a class symbol with the name defined in class node, add it to the currently open container node of the global symbol table using algorithm 4.1.7, and open the class symbol as described in section 4.1.4.
- `EndVisit(ClassNode& classNode)`: Close the currently open class symbol of the global symbol table as described in section 4.1.4.

- `BeginVisit(InterfaceNode& interfaceNode)`: Create an interface symbol with the name defined in the interface node, add it to the currently open container of the global symbol table using algorithm 4.1.7, and open the interface symbol as described in section 4.1.4.
- `EndVisit(InterfaceNode& interfaceNode)`: Close the currently open interface type symbol of the global symbol table as described in section 4.1.4.
- `BeginVisit(ConstructorNode& constructorNode)`: Open a function scope using constructorNode as the function node in algorithm 4.2.2. Create implicit `this` parameter and add it to the currently open container using algorithm 4.1.7.
- `EndVisit(ConstructorNode& constructorNode)`: Close the currently open function scope using algorithm 4.2.3.
- `BeginVisit(DestructorNode& destructorNode)`: Open a function scope using destructorNode as the function node in algorithm 4.2.2. Create implicit `this` parameter and add it to the currently open container using algorithm 4.1.7.
- `EndVisit(DestructorNode& destructorNode)`: Close the currently open function scope using algorithm 4.2.3.
- `BeginVisit(MemberFunctionNode& memberFunctionNode)`: Open a function scope using memberFunctionNode as the function node in algorithm 4.2.2. Create implicit `this` parameter and add it to the currently open container using algorithm 4.1.7.
- `EndVisit(MemberFunctionNode& memberFunctionNode)`: Close the currently open function scope using algorithm 4.2.3.
- `BeginVisit(ConversionFunctionNode& conversionFunctionNode)`: Open a function scope using conversionFunctionNode as the function node in algorithm 4.2.2. Create implicit `this` parameter and add it to the currently open container using algorithm 4.1.7.
- `EndVisit(ConversionFunctionNode& conversionFunctionNode)`: Close the currently open function scope using algorithm 4.2.3.
- `BeginVisit(StaticConstructorNode& staticConstructorNode)`: Open a function scope using staticConstructorNode as the function node in algorithm 4.2.2.
- `EndVisit(StaticConstructorNode& staticConstructorNode)`: Close the currently open function scope using algorithm 4.2.3.
- `BeginVisit(EnumTypeNode& enumTypeNode)`: Create an enumerated type symbol with name defined in enumTypeNode and add it to the currently open container using algorithm 4.1.7. Open the enumerated type symbol as described in section 4.1.4.
- `EndVisit(EnumTypeNode& enumTypeNode)`: Close the currently open enumerated type symbol of the global symbol table as described in section 4.1.4.
- `Visit(EnumConstantNode& enumConstantNode)`: Create an enumerated constant symbol with name defined in enumConstantNode and add it to the currently open container using algorithm 4.1.7.

- `Visit(TypedefNode& typedefNode)`: Create a typedef symbol with name defined in typedefNode and add it to the currently open container using algorithm 4.1.7.
- `BeginVisit(FunctionNode& functionNode)`: Open a function scope using the function node in algorithm 4.2.2.
- `EndVisit(FunctionNode& functionNode)`: Close the currently open function scope using algorithm 4.2.3.
- `BeginVisit(DelegateNode& delegateNode)`: Create a delegate type symbol with name defined in delegateNode and add it to the currently open container using algorithm 4.1.7. Open the delegate type symbol as described in section 4.1.4.
- `EndVisit(DelegateNode& delegateNode)`: Close the currently open delegate type symbol of the global symbol table as described in section 4.1.4.
- `BeginVisit(ClassDelegateNode& classDelegateNode)`: Create a delegate type symbol with name defined in classDelegateNode and add it to the currently open container using algorithm 4.1.7. Open the class delegate type symbol as described in section 4.1.4.
- `EndVisit(ClassDelegateNode& classDelegateNode)`: Close the currently open class delegate type symbol of the global symbol table as described in section 4.1.4.
- `Visit(ConstantNode& constantNode)`: Create a constant symbol with name defined in constantNode and add it to the currently open container using algorithm 4.1.7.
- `Visit(ParameterNode& parameterNode)`: Create a parameter symbol with name defined in parameterNode and add it to the currently open container using algorithm 4.1.7.
- `Visit(TemplateParameterNode& templateParameterNode)`: Create a type parameter symbol with name defined in templateParameterNode and add it to the currently open container using algorithm 4.1.7.
- `Visit(MemberVariableNode& memberVariableNode)`: Create a member variable symbol with name defined in memberVariableNode and add it to the currently open container using algorithm 4.1.7.
- `BeginVisit(CompoundStatementNode& compoundStatementNode)`: Open a declaration scope using using compoundStatementNode as the statementNode in algorithm 4.2.4.
- `EndVisit(CompoundStatementNode& compoundStatementNode)`: Close the declaration block symbol using algorithm 4.2.5.
- `BeginVisit(RangeForStatementNode& rangeForStatementNode)`: Open a declaration scope using using rangeForStatementNode as the statementNode in algorithm 4.2.4.
- `EndVisit(RangeForStatementNode& rangeForStatementNode)`: Close the declaration block symbol using algorithm 4.2.5.
- `BeginVisit(ForStatementNode& forStatementNode)`: Open a declaration scope using using forStatementNode as the statementNode in algorithm 4.2.4.

- `EndVisit(ForStatementNode& forStatementNode)`: Close the declaration block symbol using algorithm 4.2.5.
- `Visit(ConstructionStatementNode& constructionStatementNode)`: Create a local variable symbol with name defined in constructionStatementNode and add it to the currently open container using algorithm 4.1.7.
- `Visit(TypedefStatementNode& typedefStatementNode)`: Create a typedef symbol with name defined in typedefStatementNode and add it to the currently open container using algorithm 4.1.7.
- `Visit(ConceptNode& conceptNode)`: Create a concept symbol with name defined in concept node and set its group name to the group name defined in concept node. Add it to the currently open container using algorithm 4.1.7.

4.3 Example

Example 4.3.1. Consider the following Cmajor source code file.

```
1  public enum TrafficLight
2  {
3      green, yellow, red
4  }
5
6  namespace Alpha.Beta
7  {
8      public class Gamma
9      {
10         public void Foo(int x)
11         {
12             int v = 0;
13         }
14         public void Foo(double y)
15         {
16             int v = 0;
17         }
18         public void Bar(bool b)
19         {
20             int v = 0;
21         }
22         private int m;
23     }
24
25     public void Delta(bool epsilon)
26     {
27         int v = 0;
28     }
29 }
```

The following abstract syntax tree is generated while parsing the previous source code file:

```
CompileUnitNode
    NamespaceNode()
        EnumTypeNode(TrafficLight)
            EnumConstantNode(green)
            EnumConstantNode(yellow)
            EnumConstantNode(red)
        NamespaceNode(Alpha.Beta)
            ClassNode(Gamma)
                FunctionNode(Foo)
                    ParameterNode(x)
                    CompoundStatementNode
                        ConstructionStatementNode(v)
                FunctionNode(Foo)
                    ParameterNode(y)
                    CompoundStatementNode
                        ConstructionStatementNode(v)
                FunctionNode(Bar)
                    ParameterNode(b)
                    CompoundStatementNode
                        ConstructionStatementNode(v)
                MemberVariableNode(m)
            FunctionNode(Delta)
                ParameterNode(epsilon)
                CompoundStatementNode
                    ConstructionStatementNode(v)
```

The following symbol table is constructed while iterating through the previous abstract syntax tree:

```
NamespaceSymbol()
    EnumTypeSymbol(TrafficLight)
        EnumConstantSymbol(green)
        EnumConstantSymbol(yellow)
        EnumConstantSymbol(red)
    NamespaceSymbol(Alpha)
        NamespaceSymbol(Beta)
            ClassTypeSymbol(Gamma)
                FunctionGroupSymbol(Foo)
                    FunctionSymbol(Foo)
                        ParameterSymbol(this)
                        ParameterSymbol(x)
                        DeclarationBlock
                            LocalVariableSymbol(v)
                    FunctionSymbol(Foo)
                        ParameterSymbol(this)
                        ParameterSymbol(y)
                        DeclarationBlock
                            LocalVariableSymbol(v)
                FunctionGroupSymbol(Bar)
                    FunctionSymbol(Bar)
                        ParameterSymbol(this)
                        ParameterSymbol(b)
                        DeclarationBlock
                            LocalVariableSymbol(v)
                MemberVariableSymbol(m)
            FunctionGroupSymbol(Delta)
                FunctionSymbol(Delta)
                    ParameterSymbol(epsilon)
                    DeclarationBlock
                        LocalVariableSymbol(v)
```

Chapter 5

Type Repository

The type repository keeps a mapping from *type identifiers* to type symbols. A type identifier is a sixteen-byte integer that uniquely identifies a type symbol.

5.1 Computing the Type Identifier for a Type Symbol

In the following sections we show how to compute a type identifier for each kind of type symbol.

5.1.1 Type Identifiers for Basic Type Symbols

The following table shows the type identifiers for basic type symbols:

Basic Type Symbol	**Type Identifier**
BoolTypeSymbol	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
CharTypeSymbol	02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
WCharTypeSymbol	03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
UCharTypeSymbol	04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
VoidTypeSymbol	05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
SByteTypeSymbol	06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
TypeSymbol	07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ShortTypeSymbo	08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
UShortTypeSymbol	09 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IntTypeSymbol	0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
UIntTypeSymbol	0B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
LongTypeSymbol	0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ULongTypeSymbol	0D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
FloatTypeSymbol	0E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
DoubleTypeSymbol	0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
NullPtrTypeSymbol	10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

5.1.2 Type Identifiers for Class and Interface Type Symbols

A type identifier for a class or interface type symbol consists of two parts. The first eight bytes is formed by two four-byte pseudorandom numbers generated using Mersenne Twister pseudorandom number generator (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html). The rest eight bytes is formed by a serial number of the class or interface.

5.1.3 Type Identifiers for Class Template Specialization Symbols

Definition 5.1.1. Let $C < A_1, \ldots, A_n >$ be a class template specialization. In that expression, C is called the *primary class type* of that class template specialization, and $A_1, \ldots, A_n$ are called the *type arguments* of that class template specialization.

The type identifier for a class template specialization symbol is computed using the following algorithm:

Algorithm 5.1.1. Computing the Type Identifier for a Class Template Specialization Symbol.

1. Let id be the type identifier of the primary class type of the class template specialization. Let n be the number of type argument type symbols of the class template specialization.
2. For $i = 0, \ldots, n - 1$:
   (a) Let a be the type identifier of the type argument i of the class template specialization.
   (b) Let p be $(i + 8)\%16$.
   (c) Let r be a rotated by p byte positions right.
   (d) Assign id **xor** r to id.
3. The type identifier of the class template specialization is id.

5.1.4 Type Identifiers for Delegate, Class Delegate and Enumerated Type Symbols

A type identifier for delegate, class delegate and enumerated type symbols is formed by four four-byte pseudorandom numbers generated using Mersenne Twister pseudorandom number generator.

5.1.5 Type Identifiers for Derived Type Symbols

A derived type information consists of a *base type*, *derivations* and *array dimensions*. We need to encode each derivation of a derived type in order to compuer a type identifier for a derived type.

The following table shows derivation code encodings for derivations:

Derivation Symbol	Derivation	Derivation Code
`const`	const	1
`&`	reference	2
`&&`	rvalue reference	3
`*`	pointer	4
`(`	left parenthesis	5
`)`	right parenthesis	6
`[]`	array	7

The type identifiers for a derived type symbol is computed using the following algorithm:

Algorithm 5.1.2. Computing the Type Identifier for a Derived Type Symbol.

1. Let id be the type identifier for the base type of the derived type. Let m be the number of derivations of the derived type. Let n be the number of array dimensions of the derived type.
2. For $i = 0, \ldots, m - 1$:
   (a) Let c the derivation code of i'th derivation of the derived type.
   (b) Let d be 1 shifted left by c bit positions.
   (c) Set the $id[i + 1]$ to $id[i + 1]$ **xor** d.
3. For $j = 0, \ldots, n - 1$:
   (a) Let a be the j'th array dimension of the derived type.
   (b) Let $d0$ be a shifted right by 24 bit positions and ANDed by 255. Let $d1$ be a shifted right by 16 bit positions and ANDed by 255. Let $d2$ be a shifted right by 8 bit positions and ANDed by 255. Let $d3$ be a ANDed by 255.
   (c) Set $id[5 + j]$ to $id[5 + j]$ **xor** $d0$. Set $id[6 + j]$ to $id[6 + j]$ **xor** $d1$. Set $id[7 + j]$ to $id[7 + j]$ **xor** $d2$. Set $id[8 + j]$ to $id[8 + j]$ **xor** $d3$.
4. The type identifier of the derived type is id.

5.2 Adding Type Symbols to the Type Repository

Each type symbol contains a type identifier. Adding a type to the type repository is done simply by adding a mapping from the type identifier of the type symbol to the type symbol itself to the *typeid* $\rightarrow$ *symbol* mappings of the type repository.

5.3 Getting a Type Symbol from the Type Repository

Getting a type symbol from the type repository by a type identifier is done by searching the *typeid* $\rightarrow$ *symbol* mappings using the type identifier. If the type identifier is found from the mappings, the corresponding type symbol is returned. Otherwise null is returned.

5.4 Making Type Symbols

Definition 5.4.1. Base Type Symbol. Let T be a type symbol. If T is a derived type symbol (`DerivedTypeSymbol`), the *base type* of T is type T without the derivations and array dimensions. Otherwise the base type of T is type T.

The following algorithms are used to make type symbols.

Algorithm 5.4.1. Making a Derived Type Symbol. The algorithm returns either an existing derived type symbol, if it is found, or creates a new derived type symbol and returns it, if it does not exist.

1. Let b be a base type, d be list of derivations and a be a list of array dimensions.
2. Let id be a type identifier computed using algorithm 5.1.2 for b, d and a.
3. If id is found in $typeid \rightarrow symbol$ mappings of the type repository, return the type symbol found.
4. Otherwise create a new derived type symbol with type identifier id, base type b, list of derivations d, list of array dimensions a. Insert it in the $typeid \rightarrow symbol$ mappings of the type repository, and return it.

Algorithm 5.4.2. Making a Pointer Type Symbol.

1. Let b be a base type symbol.
2. Make a derived type symbol using algorithm 5.4.1 with d being a list of derivations consisting a pointer derivation, and a being empty list of array dimensions, and return it.

Algorithm 5.4.3. Making a Reference Type Symbol.

1. Let b be a base type symbol.
2. Make a derived type symbol using algorithm 5.4.1 with d being a list of derivations consisting a reference derivation, and a being empty list of array dimensions, and return it.

Algorithm 5.4.4. Making an RValue Reference Type Symbol.

1. Let b be a base type symbol.
2. Make a derived type symbol using algorithm 5.4.1 with d being a list of derivations consisting an rvalue reference derivation, and a being empty list of array dimensions, and return it.

Algorithm 5.4.5. Making a Const Reference Type Symbol.

1. Let b be a base type symbol.
2. Make a derived type symbol using algorithm 5.4.1 with d being a list of derivations consisting a const derivation and a reference derivation, and a being empty list of array dimensions, and return it.

Algorithm 5.4.6. Making a Const Pointer Type Symbol.

1. Let b be a base type symbol.
2. Make a derived type symbol using algorithm 5.4.1 with d being a list of derivations consisting a const derivation and a pointer derivation, and a being empty list of array dimensions, and return it.

Algorithm 5.4.7. Making a Class Template Specialization Type Symbol.

1. Let C be a primary type of the class template specialization and $A_1, \ldots, A_n$ be the type arguments of the class template specialization.
2. Let id be a type identifier computed using algorithm 5.1.1 for C and $A_1, \ldots, A_n$.
3. If id is found in $typeid \rightarrow symbol$ mappings of the type repository, return the type symbol found.
4. Otherwise create a new class template specialization type symbol (called `TemplateTypeSymbol` in code) with type identifier id, primary type C and type arguments $A_1, \ldots, A_n$. Insert it in the $typeid \rightarrow symbol$ mappings of the type repository, and return it.

Algorithm 5.4.8. Making a Plain Type Symbol. The algorithm returns a plain type for a type symbol T. Informally, the plain type is a type without any const, reference, or rvalue reference derivations. Pointer derivations are saved in a plain type though.

1. Let T be a type symbol.
2. If $T =$ `U&` for some type U, return U.
3. Otherwise, if $T =$ `const U&` for some type U, return U.
4. Otherwise, if $T =$ `const U` for some type U, return U.
5. Otherwise, if $T =$ `U&&` for some type U, return U.
6. Otherwise, return T.

Chapter 6

Static Evaluator

The static evaluator evaluates constant expressions such as constant expressions as values of constants, constant expressions in case statements and constant expressions as values of enumeration constants. In other words it evaluates anything that must be evaluated statically at compile time.

6.1 Evaluation Stack and Value Classes

The static evaluator has a stack of values called the *evaluation stack* that holds intermediate and final results of evaluation. The values are instances of classes derived from an abstract base class named `Value`. Here's the value class hierarchy:

```
Value
    BoolValue
    CharValue
    WCharValue
    UCharValue
    SByteValue
    ByteValue
    ShortValue
    UShortValue
    IntValue
    UIntValue
    LongValue
    ULongValue
    FloatValue
    DoubleValue
    NullValue
    StringValue
    ScopedValue
```

6.2 Operand Types and Value Types

Associated with `Value` classes there is a C++ type called `OperandType` and a Cmajor type called `ValueType`. The following table shows the operand types and value types for `Value` classes:

Value Class	OperandType	ValueType
BoolValue	**bool**	**bool**
CharValue	**char**	**char**
WCharValue	**uint16_t**	**wchar**
UCharValue	**uint32_t**	**uchar**
SByteValue	**int8_t**	**sbyte**
ByteValue	**uint8_t**	**byte**
ShortValue	**int16_t**	**short**
UShortValue	**uint16_t**	**ushort**
IntValue	**int32_t**	**int**
UIntValue	**uint32_t**	**uint**
LongValue	**int64_t**	**long**
ULongValue	**uint64_t**	**ulong**
FloatValue	**float**	**float**
DoubleValue	**double**	**double**

Each `Value` class contains a value whose type is its associated `OperandType`.

6.3 Evaluating Unary Expressions

First we go through how the static evaluator evaluates unary expressions.

6.3.1 Unary Operator Functions

The static evaluator has a *unary operator function* for each supported unary operator. The unary operator function is a function template that delegates the evaluation to a C++ function object. The following table shows supported unary operators, their corresponding unary operator functions, and C++ function objects that are used in evaluating unary expressions:

Unary Operator	Unary Operator Function	C++ Function Object
`~`	`Complement<ValueT>`	`bit_not<ValueT::OperandType>`
`+`	`UnaryPlus<ValueT>`	`identity<ValueT::OperandType>`
`-`	`UnaryMinus<ValueT>`	`std::negate<ValueT::OperandType>`
`!`	`Not<ValueT>`	`std::logical_not<ValueT::OperandType>`

6.3.2 Unary Expression Evaluation Algorithm

Here's the unary expression evaluation algorithm:

Algorithm 6.3.1. Evaluation a Unary Expression. Inputs to this algorithm are:

- the target value type,
- evaluation stack,
- unary operator function `f`,
- whether to perform cast.

1. Pop the operand from the evaluation stack.
2. Let *subjectType* be the value type of the operand.
3. If the target value type is wider than *subjectType*, let *operationType* be target value type. Otherwise let *operationType* be *subjectType*.
4. Convert the operand to *operationType* type possibly performing a cast if requested. As usual in Cmajor, conversions that promote a value to a value of wider type are performed implicitly, whereas conversions to a narrower type require a cast.
5. Call the unary operator function `f<operationType>` using the converted operand as an argument.
6. Push the result to the evaluation stack.

6.4 Evaluating Binary Expressions

Next we go through how the static evaluator evaluates binary expressions.

6.4.1 Common Type

In order to evaluate a binary expression, one needs to have a *common type* to which both operands of the binary operator are converted before evaluation. Given two value types, one can compute their common type as follows:

Algorithm 6.4.1. Computing the Common Type of Two Value Types. Common type gives the smallest value type for the left and the right value type that is large enough to hold a value of the left type and a value of the right type. Given the left and the right type, common type returns a type according to the following table:

Left Type	Right Type	Common Type
bool	**bool**	**bool**
bool	**char**	
bool	**wchar**	
bool	**uchar**	

bool	**sbyte**	
bool	**byte**	
bool	**short**	
bool	**ushort**	
bool	**int**	
bool	**uint**	
bool	**long**	
bool	**ulong**	
bool	**float**	
bool	**double**	
bool	**nullPtrType**	
bool	**string**	
char	**bool**	
char	**char**	**char**
char	**wchar**	**wchar**
char	**uchar**	**uchar**
char	**sbyte**	
char	**byte**	
char	**short**	
char	**ushort**	
char	**int**	
char	**uint**	
char	**long**	
char	**ulong**	
char	**float**	
char	**double**	
char	**nullPtrType**	
char	**string**	
wchar	**bool**	
wchar	**char**	**wchar**
wchar	**wchar**	**wchar**
wchar	**uchar**	**uchar**
wchar	**sbyte**	
wchar	**byte**	
wchar	**short**	
wchar	**ushort**	
wchar	**int**	
wchar	**uint**	
wchar	**long**	
wchar	**ulong**	
wchar	**float**	
wchar	**double**	

wchar	**nullPtrType**	
wchar	**string**	
uchar	**bool**	
uchar	**char**	**uchar**
uchar	**wchar**	**uchar**
uchar	**uchar**	**uchar**
uchar	**sbyte**	
uchar	**byte**	
uchar	**short**	
uchar	**ushort**	
uchar	**int**	
uchar	**uint**	
uchar	**long**	
uchar	**ulong**	
uchar	**float**	
uchar	**double**	
uchar	**nullPtrType**	
uchar	**string**	
sbyte	**bool**	
sbyte	**char**	
sbyte	**wchar**	
sbyte	**uchar**	
sbyte	**sbyte**	**sbyte**
sbyte	**byte**	**short**
sbyte	**short**	**short**
sbyte	**ushort**	**int**
sbyte	**int**	**int**
sbyte	**uint**	**long**
sbyte	**long**	**long**
sbyte	**ulong**	
sbyte	**float**	**float**
sbyte	**double**	**double**
sbyte	**nullPtrType**	
sbyte	**string**	
byte	**bool**	
byte	**char**	
byte	**wchar**	
byte	**uchar**	
byte	**sbyte**	**short**
byte	**byte**	**byte**
byte	**short**	**short**
byte	**ushort**	**ushort**

byte	**int**	**int**
byte	**uint**	**uint**
byte	**long**	**long**
byte	**ulong**	**ulong**
byte	**float**	**float**
byte	**double**	**double**
byte	**nullPtrType**	
byte	**string**	
short	**bool**	
short	**char**	
short	**wchar**	
short	**uchar**	
short	**sbyte**	**short**
short	**byte**	**short**
short	**short**	**short**
short	**ushort**	**int**
short	**int**	**int**
short	**uint**	**long**
short	**long**	**long**
short	**ulong**	
short	**float**	**float**
short	**double**	**double**
short	**nullPtrType**	
short	**string**	
ushort	**bool**	
ushort	**char**	
ushort	**wchar**	
ushort	**uchar**	
ushort	**sbyte**	**int**
ushort	**byte**	**ushort**
ushort	**short**	**int**
ushort	**ushort**	**ushort**
ushort	**int**	**int**
ushort	**uint**	**uint**
ushort	**long**	**long**
ushort	**ulong**	**ulong**
ushort	**float**	**float**
ushort	**double**	**double**
ushort	**nullPtrType**	
ushort	**string**	
int	**bool**	
int	**char**	

int	**wchar**	
int	**uchar**	
int	**sbyte**	**int**
int	**byte**	**int**
int	**short**	**int**
int	**ushort**	**int**
int	**int**	**int**
int	**uint**	**long**
int	**long**	**long**
int	**ulong**	
int	**float**	**float**
int	**double**	**double**
int	**nullPtrType**	
int	**string**	
uint	**bool**	
uint	**char**	
uint	**wchar**	
uint	**uchar**	
uint	**sbyte**	**long**
uint	**byte**	**uint**
uint	**short**	**long**
uint	**ushort**	**uint**
uint	**int**	**long**
uint	**uint**	**uint**
uint	**long**	**long**
uint	**ulong**	**ulong**
uint	**float**	**float**
uint	**double**	**double**
uint	**nullPtrType**	
uint	**string**	
long	**bool**	
long	**char**	
long	**wchar**	
long	**uchar**	
long	**sbyte**	**long**
long	**byte**	**long**
long	**short**	**long**
long	**ushort**	**long**
long	**int**	**long**
long	**uint**	**long**
long	**long**	**long**
long	**ulong**	

long	**float**	**float**
long	**double**	**double**
long	**nullPtrType**	
long	**string**	
ulong	**bool**	
ulong	**char**	
ulong	**wchar**	
ulong	**uchar**	
ulong	**sbyte**	
ulong	**byte**	**ulong**
ulong	**short**	
ulong	**ushort**	**ulong**
ulong	**int**	
ulong	**uint**	**ulong**
ulong	**long**	
ulong	**ulong**	**ulong**
ulong	**float**	**float**
ulong	**double**	**double**
ulong	**nullPtrType**	
ulong	**string**	
float	**bool**	
float	**char**	
float	**wchar**	
float	**uchar**	
float	**sbyte**	**float**
float	**byte**	**float**
float	**short**	**float**
float	**ushort**	**float**
float	**int**	**float**
float	**uint**	**float**
float	**long**	**float**
float	**ulong**	**float**
float	**float**	**float**
float	**double**	**double**
float	**nullPtrType**	
float	**string**	
double	**bool**	
double	**char**	
double	**wchar**	
double	**uchar**	
double	**sbyte**	**double**
double	**byte**	**double**

double	**short**	**double**
double	**ushort**	**double**
double	**int**	**double**
double	**uint**	**double**
double	**long**	**double**
double	**ulong**	**double**
double	**float**	**double**
double	**double**	**double**
double	**nullPtrType**	
double	**string**	
nullPtrType	**bool**	
nullPtrType	**char**	
nullPtrType	**wchar**	
nullPtrType	**uchar**	
nullPtrType	**sbyte**	
nullPtrType	**byte**	
nullPtrType	**short**	
nullPtrType	**ushort**	
nullPtrType	**int**	
nullPtrType	**uint**	
nullPtrType	**long**	
nullPtrType	**ulong**	
nullPtrType	**float**	
nullPtrType	**double**	
nullPtrType	**nullPtrType**	**nullPtrType**
nullPtrType	**string**	
string	**bool**	
string	**char**	
string	**wchar**	
string	**uchar**	
string	**sbyte**	
string	**byte**	
string	**short**	
string	**ushort**	
string	**int**	
string	**uint**	
string	**long**	
string	**ulong**	
string	**float**	
string	**double**	
string	**nullPtrType**	
string	**string**	**string**

6.4.2 Binary Operator Functions

The static evaluator has a *binary operator function* for each supported binary operator. The binary operator function is a function template that delegates the evaluation to a C++ function function object. The following table shows supported binary operators, their corresponding binary operator functions, and C++ function objects that are used in evaluating binary expressions:

Binary Operator	Binary Operator Function	C++ Function Object
&&	Conjunction<ValueT>	std::logical_and<ValueT::OperandType>
||	Disjunction<ValueT>	std::logical_or<ValueT::OperandType>
^	BitXor<ValueT>	std::bit_xor<ValueT::OperandType>
|	BitOr<ValueT>	std::bit_or<ValueT::OperandType>
&	BitAnd<ValueT>	std::bit_and<ValueT::OperandType>
%	Rem<ValueT>	std::modulus<ValueT::OperandType>
/	Div<ValueT>	std::divides<ValueT::OperandType>
*	Mul<ValueT>	std::multiplies<ValueT::OperandType>
-	Sub<ValueT>	std::minus<ValueT::OperandType>
+	Add<ValueT>	std::plus<ValueT::OperandType>
>>	ShiftRight<ValueT>	shiftRightFun<ValueT::OperandType>
<<	ShiftLeft<ValueT>	shiftLeftFun<ValueT::OperandType>
==	Equal<ValueT>	std::equal_to<ValueT::OperandType>
!=	NotEqual<ValueT>	std::not_equal_to<ValueT::OperandType>
<	Less<ValueT>	std::less<ValueT::OperandType>
>	Greater<ValueT>	std::greater<ValueT::OperandType>
<=	LessOrEqual<ValueT>	std::less_equal<ValueT::OperandType>
>=	GreaterOrEqual<ValueT>	std::greater_equal<ValueT::OperandType>

6.4.3 Binary Expression Evaluation Algorithm

Finally here's the binary expression evaluation algorithm:

Algorithm 6.4.2. Evaluating a Binary Expression. Inputs to this algorithm are:

- the target value type
- evaluation stack
- a binary operator function `f`,
- whether to perform cast.

1. Pop the right operand from the evaluation stack.
2. Pop the left operand from the evaluation stack.

3. Let *leftType* be the value type of the left operand. Let *rightType* be the value type of the right operand. Let *commonType* be the common value type computed using algorithm 6.4.1 for *leftType* and *rightType*.

4. If the target value type is wider than *commonType*, let *operationType* be target value type. Otherwise let *operationType* be *commonType*.

5. Convert the left and right operands to *operationType* type possibly performing a cast if requested.

6. Call the binary operator function `f<operationType>` using converted left and right operands as arguments.

7. Push the result to the evaluation stack.

6.5 Evaluating the Value Associated with a Symbol

Evaluation of the value associated with a symbol is done using the following algorithm:

Algorithm 6.5.1. Evaluating the Value Associated with a Symbol. Inputs to this algorithm are:

- a symbol and
- an evaluation stack.

1. If the symbol is a container symbol, create a new `ScopedValue` containing the container symbol, and push it to the evaluation stack.

2. Otherwise, if the symbol is a constant symbol, evaluate the expression of the constant symbol using algorithm 6.6.1, and push it to the evaluation stack.

3. Otherwise, if the symbol is an enumerated constant symbol, evaluate the expression of the enumeration constant symbol using algorithm 6.6.1, and push it to the evaluation stack.

4. Otherwise, throw an exception.

6.6 Evaluation of a Constant Expression

The main algorithm of this component is the evaluation of a constant expression:

Algorithm 6.6.1. Evaluating a Constant Expression. The inputs to this algorithm are:

- a constant expression represented as an abstract syntax tree node,
- a target value type, i.e. the type to which the evaluated result is finally converted,
- whether a cast is performed,
- a container scope and file scopes (see section 8.1) for symbol lookup.

The algorithm returns an instance of a class derived from `Value` class that contains the evaluated result.

The algorithm creates an instance of a static evaluator that is an abstract syntax tree visitor, and calls the `Accept` member function of the given abstract syntax tree node by giving the static evaluator instance as an argument. As a result of the visitation, the evaluated value will be in the top of the evaluation stack. Finally the evaluated value is popped off from the evaluation stack, converted to the required target type, and returned.

The static evaluator overrides the following visiting points:

- `Visit(BooleanLiteralNode& booleanLiteralNode)`: Create an instance of a `BoolValue` containing the value from the `booleanLiteralNode`, and push it to the evaluation stack.
- `Visit(SByteLiteralNode& sbyteLiteralNode)`: Create an instance of a `SByteValue` containing the value from the `sbyteLiteralNode`, and push it to the evaluation stack.
- `Visit(ByteLiteralNode& byteLiteralNode)`: Create an instance of a `ByteValue` containing the value from the `byteLiteralNode`, and push it to the evaluation stack.
- `Visit(ShortLiteralNode& shortLiteralNode)`: Create an instance of a `ShortValue` containing the value from the `shortLiteralNode`, and push it to the evaluation stack.
- `Visit(UShortLiteralNode& ushortLiteralNode)`: Create an instance of a `UShortValue` containing the value from the `ushortLiteralNode`, and push it to the evaluation stack.
- `Visit(IntLiteralNode& intLiteralNode)`: Create an instance of a `IntValue` containing the value from the `intLiteralNode`, and push it to the evaluation stack.
- `Visit(UIntLiteralNode& uintLiteralNode)`: Create an instance of a `UIntValue` containing the value from the `uintLiteralNode`, and push it to the evaluation stack.
- `Visit(LongLiteralNode& longLiteralNode)`: Create an instance of a `LongValue` containing the value from the `longLiteralNode`, and push it to the evaluation stack.
- `Visit(ULongLiteralNode& ulongLiteralNode)`: Create an instance of a `ULongValue` containing the value from the `ulongLiteralNode`, and push it to the evaluation stack.
- `Visit(FloatLiteralNode& floatLiteralNode)`: Create an instance of a `FloatValue` containing the value from the `floatLiteralNode`, and push it to the evaluation stack.
- `Visit(DoubleLiteralNode& doubleLiteralNode)`: Create an instance of a `DoubleValue` containing the value from the `doubleLiteralNode`, and push it to the evaluation stack.
- `Visit(CharLiteralNode& charLiteralNode)`: Create an instance of a `CharValue` containing the value from the `charLiteralNode`, and push it to the evaluation stack.
- `EndVisit(DisjunctionNode& disjunctionNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Disjunction` binary operator function, and cast as arguments.
- `EndVisit(ConjunctionNode& conjunctionNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Conjunction` binary operator function, and cast as arguments.

- `EndVisit(BitOrNode& bitOrNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `BitOr` binary operator function, and cast as arguments.
- `EndVisit(BitXorNode& bitXorNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `BitXor` binary operator function, and cast as arguments.
- `EndVisit(BitAndNode& bitAndNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `BitAnd` binary operator function, and cast as arguments.
- `EndVisit(EqualNode& equalNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Equal` binary operator function, and cast as arguments.
- `EndVisit(NotEqualNode& notEqualNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `NotEqual` binary operator function, and cast as arguments.
- `EndVisit(LessNode& lessNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Less` binary operator function, and cast as arguments.
- `EndVisit(GreaterNode& greaterNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Greater` binary operator function, and cast as arguments.
- `EndVisit(LessOrEqualNode& lessOrEqualNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `LessOrEqual` binary operator function, and cast as arguments.
- `EndVisit(GreaterOrEqualNode& greaterOrEqualNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `GreaterOrEqual` binary operator function, and cast as arguments.
- `EndVisit(ShiftLeftNode& shiftLeftNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `ShiftLeft` binary operator function, and cast as arguments.
- `EndVisit(ShiftRightNode& shiftRightNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `ShiftRight` binary operator function, and cast as arguments.
- `EndVisit(AddNode& addNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Add` binary operator function, and cast as arguments.
- `EndVisit(SubNode& subNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Sub` binary operator function, and cast as arguments.

- `EndVisit(MulNode& mulNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Mul` binary operator function, and cast as arguments.
- `EndVisit(DivNode& divNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Div` binary operator function, and cast as arguments.
- `EndVisit(RemNode& remNode)`: Evaluate a binary expression by calling algorithm 6.4.2 and giving the target value type, evaluation stack, `Rem` binary operator function, and cast as arguments.
- `EndVisit(UnaryPlusNode& unaryPlusNode)`: Evaluate a unary expression by calling algorithm 6.3.1 and giving the target value type, evaluation stack, `UnaryPlus` unary operator function, and cast as arguments.
- `EndVisit(UnaryMinusNode& unaryMinusNode)`: Evaluate a unary expression by calling algorithm 6.3.1 and giving the target value type, evaluation stack, `UnaryMinus` unary operator function, and cast as arguments.
- `EndVisit(NotNode& notNode)`: Evaluate a unary expression by calling algorithm 6.3.1 and giving the target value type, evaluation stack, `Not` unary operator function, and cast as arguments.
- `EndVisit(ComplementNode& complementNode)`: Evaluate a unary expression by calling algorithm 6.3.1 and giving the target value type, evaluation stack, `Complement` unary operator function, and cast as arguments.
- `EndVisit(DotNode& dotNode)`: Pop a value from the evaluation stack. If the value is of `ScopedValue` type, lookup an identifier defined in the `dotNode` from the container scope of the container symbol defined in the `ScopedValue` using lookup algorithms in section 4.1.3. If a symbol is found, use algorithm 6.5.1 to evaluate the value associated with the symbol. Otherwise throw an exception.
- `Visit(CastNode& castNode)`: Let e be the target type expression represented as an abstract syntax tree node that is contained by the `castNode`. Use the type resolver to resolve the type from e by using algorithm **??**. Let t be the resolved type symbol. Let v be the value type for t. Evaluate the constant expression defined in the `castNode` by calling this algorithm recursively with v as the target type and cast to be **true**. Push the evaluated value to the evaluation stack.
- `Visit(IdentifierNode& identifierNode)`: Lookup the identifier defined in the `identifierNode` from the container scope and file scopes (see section 8.1) using lookup algorithms in section 4.1.3. If a symbol is found, use algorithm 6.5.1 to evaluate the value associated with the symbol. Otherwise throw an exception.

6.7 Example

Let's go through evaluation the value of two constants defined in the following listing:

```
1 public const int a = 2;
2 public const int b = 2 * (a + 3);
```

When the source file containing previous definitions is parsed, the following abstract syntax tree is generated:

```
CompileUnitNode
    NamespaceNode()
        ConstantNode
            IntNode
            IdentifierNode(a)
            SByteLiteralNode(2)
        ConstantNode
            IntNode
            IdentifierNode(b)
            MulNode
                SByteLiteralNode(2)
                AddNode
                    IdentifierNode(a)
                    SByteLiteralNode(3)
```

6.7.1 Evaluation of Constant a

For the constant a, the inputs for algorithm 6.6.1 are:

- constant expression node: `SByteLiteraNode(2)`
- target value type: **int**
- container scope: the global scope.

The following steps are executed:

1. `Visit(SByteLiteralNode& sbyteLiteralNode)` is called. This causes an `SByteValue` with value 2 to be created and pushed to the evaluation stack.
2. The `SByteValue` with value 2 is popped from the evaluation stack, converted to `IntValue` and returned.

6.7.2 Evaluation of Constant b

For the constant b, the inputs for algorithm 6.6.1 are:

- constant expression nodes:

```
MulNode
    SByteLiteralNode(2)
    AddNode
        IdentifierNode(a)
        SByteLiteralNode(3)
```

- target value type: **int**
- container scope: the global scope.

The following steps are executed:

1. `Visit(SByteLiteralNode& sbyteLiteralNode)` is called. This causes an `SByteValue` with value 2 to be created and pushed to the evaluation stack. Now the contents of the evaluation stack is: `SByteValue(2)`.
2. `Visit(IdentifierNode& identifierNode)` is called.
   (a) Identifier a is looked up from the global scope.
   (b) Constant symbol a is found from the global scope.
   (c) Algorithm 6.5.1 is executed with symbol a:
      i. Symbol a is a constant symbol, so its value `IntValue(2)` is pushed to the evaluation stack.

   Now the contents of the evaluation stack is: `SByteValue(2),IntValue(2)`.
3. `Visit(SByteLiteralNode& sbyteLiteralNode)` is called. This causes an `SByteValue` with value 3 to be created and pushed to the evaluation stack. Now the contents of the evaluation stack is: `SByteValue(2),IntValue(2),SByteValue(3)`.
4. `EndVisit(AddNode& addNode)` is called: Algorithm 6.4.2 is executed with `Add` binary operator function:
   (a) Right operand `SByteValue(3)` is popped from the evaluation stack.
   (b) Left operand `IntValue(2)` is popped from the evaluation stack.
   (c) The $leftType$ is **int**, $rightType$ is **sbyte**, $commonType$ is **int** and $operationType$ is **int**.
   (d) The right `SByteValue(3)` operand is converted to `IntValue(3)`.
   (e) `Add<IntValue>(IntValue(2), IntValue(3))` is called and `std::plus<int>()(2, 3)` is evaluated.
   (f) Result `IntValue(5)` is pushed to the evaluation stack.

   Now the contents of the evaluation stack is: `SByteValue(2),IntValue(5)`.
5. `EndVisit(MulNode& mulNode)` is called: Algorithm 6.4.2 is executed with `Mul` binary operator function:
   (a) Right operand `IntValue(5)` is popped from the evaluation stack.
   (b) Left operand `SByteValue(2)` is popped from the evaluation stack.

(c) The $leftType$ is **sbyte**, $rightType$ is **int**, $commonType$ is **int** and $operationType$ is **int**.

(d) The left `SByteValue(2)` operand is converted to `IntValue(2)`.

(e) `Mul<IntValue>(IntValue(2), IntValue(5))` is called and `std::multiplies<int>()(2, 5)` is evaluated.

(f) Result `IntValue(10)` is pushed to the evaluation stack.

6. The `IntValue` with value 10 is popped from the evaluation stack and returned.

Chapter 7

Type Resolver

The type resolver resolves a type symbol for given type expression represented as an abstract syntax tree node.

7.1 Type Symbol Hierarchy

The type symbol returned is one of the following:

```
TypeSymbol
    BasicTypeSymbol
        BoolTypeSymbol
        CharTypeSymbol
        WCharTypeSymbol
        UCharTypeSymbol
        VoidTypeSymbol
        SByteTypeSymbol
        ByteTypeSymbol
        ShortTypeSymbol
        UShortTypeSymbol
        IntTypeSymbol
        UIntTypeSymbol
        LongTypeSymbol
        ULongTypeSymbol
        FloatTypeSymbol
        DoubleTypeSymbol
        NullPtrTypeSymbol
    DerivedTypeSymbol
    EnumTypeSymbol
    ClassTypeSymbol
        TemplateTypeSymbol
    InterfaceTypeSymbol
    DelegateTypeSymbol
    ClassDelegateTypeSymbol
    TypeParameterSymbol
    NamespaceTypeSymbol
```

7.2 Type Resolving Algorithms

Resolving a type is done using the following algorithm:

Algorithm 7.2.1. Resolving a Type. Inputs to this algorithm are:

- a type expression represented as an abstract syntax tree node
- a container scope and list of file scopes (see section 8.1) for symbol lookup,
- class template repository (see section **??**).
- options: `none` or `dontThrow`.

The algorithm returns the resolved type symbol if successful. Otherwise either an exception will be thrown or null is returned depending on the options. The type resolver is implemented as an abstract syntax tree visitor.

1. The `Accept` member function of the type expression syntax tree node is called by giving the type resolver as the visitor argument.
2. As a result of visitation, the `typeSymbol` member variable holds the resolved type symbol that is returned to the caller.

The type resolver visitor overrides the following visitation points:

- `Visit(BoolNode& boolNode)`: Set `typeSymbol` to `BoolTypeSymbol`.
- `Visit(SByteNode& sbyteNode)`: Set `typeSymbol` to `SByteTypeSymbol`.
- `Visit(ByteNode& byteNode)`: Set `typeSymbol` to `ByteTypeSymbol`.
- `Visit(ShortNode& shortNode)`: Set `typeSymbol` to `ShortTypeSymbol`.
- `Visit(UShortNode& ushortNode)`: Set `typeSymbol` to `UShortTypeSymbol`.
- `Visit(IntNode& intNode)`: Set `typeSymbol` to `IntTypeSymbol`.
- `Visit(UIntNode& uintNode)`: Set `typeSymbol` to `UIntTypeSymbol`.
- `Visit(LongNode& longNode)`: Set `typeSymbol` to `LongTypeSymbol`.
- `Visit(ULongNode& ulongNode)`: Set `typeSymbol` to `ULongTypeSymbol`.
- `Visit(FloatNode& floatNode)`: Set `typeSymbol` to `FloatTypeSymbol`.
- `Visit(DoubleNode& doubleNode)`: Set `typeSymbol` to `DoubleTypeSymbol`.
- `Visit(CharNode& charNode)`: Set `typeSymbol` to `CharTypeSymbol`.
- `Visit(WCharNode& wcharNode)`: Set `typeSymbol` to `WCharTypeSymbol`.
- `Visit(UCharNode& ucharNode)`: Set `typeSymbol` to `UCharTypeSymbol`.
- `Visit(VoidNode& voidNode)`: Set `typeSymbol` to `VoidTypeSymbol`.

- `Visit(DerivedTypeExprNode& derivedTypeExprNode)`: The `DerivedTypeExprNode` contains:
  - base type expression represented as an abstract syntax tree node
  - a list of array dimensions represented as abstract syntax tree nodes
  - a list of derivations where derivation is either
    1. `const`,
    2. `reference`,
    3. `rvalue reference`, or
    4. `pointer`

  Steps for resolving a type symbol from `DerivedTypeExprNode` are:

  1. Resolve the base type from the base type expression by calling this algorithm recursively.
  2. Evaluate the array dimensions from the list of array dimension syntax tree nodes using static evaluator (algorithm 6.6.1).
  3. Make derived type symbol using type repository algorithm 5.4.1 with list of derivations, base type symbol, and array dimensions as arguments.
  4. Set `typeSymbol` to returned derived type symbol.

- `Visit(TemplateIdNode& templateIdNode)`: The `TemplateIdNode` contains:
  - an abstract syntax tree node that represents the primary class type of a class template specialization (definition 5.1.1).
  - a list of abstract syntax tree nodes that represent the type arguments of a class template specialization.

  Steps for resolving a type symbol from `TemplateIdNode` are:

  1. Resolve the primary class type symbol by calling this algorithm recursively. Let p be the resolved primary class type symbol. Let n be the number of type parameters of the primary class type symbol.
  2. Resolve the type arguments by calling this algorithm recursively. Let a be the list of type arguments resolved. Let m be the length of list a.
  3. If $m < n$ use algorithm **??** of class template repository to resolve default type arguments and append them to list a.
  4. Make a class template specialization symbol using algorithm 5.4.7 of type repository with arguments p and a.
  5. Set `typeSymbol` to returned class template specialization type symbol.

- `Visit(IdentifierNode& identifierNode)`: Steps for resolving a type symbol from `IdentifierNode` are:

  1. Lookup a symbol for identifier contained by the `identifierNode` from the container scope and file scopes using algorithms in section 4.1.3.

2. If successful, use algorithm 7.2.2 to resolve the type associated with the symbol found and set it to `typeSymbol`.
3. Otherwise either throw an exception, or set `typeSymbol` to null depending on options.

- `EndVisit(DotNode& dotNode)`: Steps for resolving a type symbol from `DotNode` are:

1. At this point `typeSymbol` should contain a type symbol that represents the left part before the dot.
2. If `typeSymbol` is `ClassTypeSymbol`, let c be the container scope of the `ClassTypeSymbol`.
3. Otherwise if `typeSymbol` is `NamespaceTypeSymbol`, let c be the container scope of the namespace symbol contained by the `NamespaceTypeSymbol`.
4. Lookup a symbol for identifier contained by the `dotNode` from the container scope c.
5. If successful, use algorithm 7.2.2 to resolve the type associated with the symbol found and set it to `typeSymbol`.
6. Otherwise either throw an exception, or set `typeSymbol` to null depending on options.

The algorithm used to resolve a type symbol associated with a symbol follows:

Algorithm 7.2.2. Resolving a Type Symbol Associated with a Symbol. Inputs to this algorithm are:

- a symbol
- options: `none` or `dontThrow`

The algorithm returns a type symbol if successful, or either throws an exception or returns null depending on options otherwise. If the symbol is:

1. a `TypeSymbol` return the type symbol itself.
2. a `TypedefSymbol` return the type symbol associated with the `TypedefSymbol`.
3. a `BoundTypeParameterSymbol` return the type symbol associated with the `BoundTypeParameterSymbol`.
4. a `NamespaceSymbol` create a `NamespaceTypeSymbol` that contains the namespace symbol and return it.
5. Otherwise either throw an exception or return null depending on options.

7.3 Example

Consider the following code:

```
1 public class Set<T, C = System.Less<T>>
2 {
3     // ...
4 }
5
6 public void foo(const Set<int>& x)
7 {
8     // ...
9 }
```

The following abstract syntax tree is generated from the code above:

```
CompileUnitNode
    NamespaceNode()
        ClassNode(Set)
            TemplateParameterNode
                IdentifierNode(T)
            TemplateParameterNode
                IdentifierNode(C)
                TemplateIdNode
                    IdentifierNode(System.Less)
                    IdentifierNode(T)
        FunctionNode
            FunctionGroupIdNode(foo)
            ParameterNode
                DerivedTypeExprNode
                    Derivation.const
                    Derivation.reference
                    TemplateIdNode
                        IdentifierNode(Set)
                        IntNode
                IdentifierNode(x)
            CompoundStatementNode
```

The following symbol table is generated from the abstract syntax tree above:

```
NamespaceSymbol()
    ClassTypeSymbol(Set)
        TypeParameterSymbol(T)
        TypeParameterSymbol(C)
    FunctionSymbol(foo)
        ParameterSymbol(x)
        DeclarationBlock
```

Here we go through the steps for resolving a type for the parameter `x` of the function `foo`. The type resolver is given the following parameters as input:

- type expression:

```
DerivedTypeExprNode
    Derivation.const
    Derivation.reference
    TemplateIdNode
        IdentifierNode(Set)
        IntNode
```

- scopes:
  - container scope: the global scope
  - file scopes: -
- class template repository
- options: `none`

1. `Visit(DerivedTypeExprNode& derivedTypeExprNode)`:
   - the base type expression is:

     ```
     TemplateIdNode
         IdentifierNode(Set)
         IntNode
     ```

   - no array dimensions.
   - derivations are: `Derivation.const` `Derivation.reference`
2. Calling algorithm 7.2.1 for base type expression:
3. `Visit(TemplateIdNode& templateIdNode)`:
   - abstract syntax tree node representing the primary class type is: `IdentifierNode(Set)`
   - abstract syntax tree nodes representing type arguments are: `IntNode`

   Calling algorithm 7.2.1 for primary class type:
4. `Visit(IdentifierNode& identifierNode)`: Looking up identifier `Set`: Symbol `ClassTypeSymbol(Set)` found.
5. Resolving the type symbol using algorithm 7.2.2 yields `ClassTypeSymbol(Set)` itself.
6. Return the type symbol `ClassTypeSymbol(Set)` as the primary class type.
7. Calling algorithm 7.2.1 for type argument `IntNode`:
8. `Visit(IntNode& intNode)`: Return `IntTypeSymbol` as a type argument.
9. The primary class type has two type parameters, but the list of type arguments contain only one type symbol, so algorithm **??** of class template repository is used to resolve the second type argument to `System.Less<int>`.

10. Algorithm 5.4.7 is used to make a class template specialization `Set<int, System.Less<int>>`.

11. Make derived type symbol using type repository algorithm 5.4.1 with list of derivations `const` and `reference` and base type `Set<int, System.Less<int>>`.

12. Finally a type symbol `const Set<int, System.Less<int>>&` is returned.

Chapter 8

Importing Namespaces, and Binding Types and Values

In the next phase of compilation we:

- import namespaces,
- define aliases for symbols,
- bind types and values to symbols,
- set access to symbols, and
- check the validity of specifiers.

We begin by defining file scopes.

8.1 File Scopes

A *file scope* consists of container scopes of imported namespaces and aliases for symbols in the header of a Cmajor source file. For example, the following Cmajor source file contains two namespace imports and two symbol aliases:

```
1 using System;                                  // imported namespace
2 using System.Collections;                      // imported namespace
3 using Str = System.String;                     // alias for type symbol
4 using StrLen = System.Support.StrLen;          // alias for function group
```

Thus the file scope for the previous Cmajor source file contains container scopes for namespaces `System` and `System.Collections`, and alias mappings from strings `Str` and `StrLen` to symbols `System.String` and `System.Support.StrLen` respectively.

Looking up a symbol for a name from a file scope is done using the following algorithm:

Algorithm 8.1.1. Lookup a Name from a File Scope.

1. Search the name from the alias mappings of the file scope.
2. If the name is found, return the mapped symbol.

3. Otherwise, lookup the name from the container scopes of the file scope.
4. If more than one symbol found, report the ambiguity by throwing an exception.
5. Otherwise, if exactly one symbol is found, return the symbol found.
6. Otherwise, no symbols found, so return null.

8.2 Binding Types and Values

The type resolver (see section 7) is used to resolve:

- types of constants,
- underlying types of enumerated types, when specified,
- types of local and member variables
- base class types and implemented interface types for class types,
- parameter and return types of functions, delegates and class delegates, and
- types of typedefs.

The static evaluator (see section 6) is used to evaluate:

- values of constants, and
- values of enumeration constants.

8.3 Setting Access to Symbols

In Cmajor one can associate one of the following access specifiers to a namespace-level or class-level object:

- **public**
- **protected**
- **internal**
- **private**

Setting access to a symbol is done using the following algorithm:

Algorithm 8.3.1. Setting Access to a Symbol. Let s be one of the symbols:

- `ClassTypeSymbol`
- `ConceptSymbol`
- `ConstantSymbol`
- `DelegateTypeSymbol`

- `ClassDelegateTypeSymbol`
- `EnumTypeSymbol`
- `FunctionSymbol`
- `InterfaceTypeSymbol`
- `MemberVariableSymbol`
- `TypedefSymbol`

Let n be the abstract syntax tree node that corresponds s. Let a be the set of access specifiers defined for n. If n is a member of a class, let *access* be **private**, otherwise let *access* be **internal**.

1. If a is {**public**}, set *access* to **public**.
2. Otherwise, if a is {**protected**}:
   (a) If n is a member of a class, set *access* to **protected**.
   (b) Otherwise throw exception `only class members can have protected access`.
3. Otherwise, if a is {**internal**}, set *access* to **internal**.
4. Otherwise, if a is {**private**}:
   (a) If n is a member of a class, set *access* to **private**.
   (b) Otherwise throw exception `only class members can have private access`.
5. Otherwise, if a is not empty, throw exception `invalid combination of access specifiers`.
6. Set access of s to *access*.

8.4 Checking the Validity of Specifiers

The following table shows possible specifiers for each symbol:[1]

Symbol	Specifiers
`ClassTypeSymbol`	**static abstract public protected private internal**
`ConceptSymbol`	**public protected private internal**
`ConstantSymbol`	**public protected private internal**
`DelegateTypeSymbol`	**nothrow throw public protected private internal**
`ClassDelegateTypeSymbol`	**nothrow throw public protected private internal**
`EnumTypeSymbol`	**public protected private internal**
`FunctionSymbol`	**static explicit extern suppress default inline cdecl nothrow throw abstract virtual override new public protected private internal**

[1] not all combination are valid

`InterfaceTypeSymbol`	**public protected private internal**
`MemberVariableSymbol`	**static public protected private internal**
`TypedefSymbol`	**public protected private internal**

Chapter 9

Function Repositories

A *function repository* is a container of certain kinds of functions. It is used for collecting viable functions for *overload resolution* and caching them per compilation unit basis. Each function repository matches its *function signatures* with the group name (see definition 4.1.1), arity (see definition 4.1.2) and arguments of searched functions. The results are collected to a list of *viable functions*.

In the start of the compilation of a compile unit, each function repository is empty. When a matching function signature is found, a function with that signature is created and inserted to the function repository. If later in compilation of the compile unit, the same signature is found again, the cached function is inserted to viable functions.

The function repositories are:

- derived type operation repository
- enumerated type operation repository
- array type operation repository
- interface type operation repository
- delegate type operation repository
- class delegate type operation repository
- synthesized class function repository

9.1 Collecting Viable Functions from Function Repositories

When searching for matching functions, the caller provides *argument information structures* for function signature matching.

An argument information structure contains:

- category of argument (`rvalue` or `lvalue`),
- type of argument, and
- whether to bind argument to an rvalue reference.

The following algorithm is used for collecting viable functions from function repositories:

Algorithm 9.1.1. Collecting Viable Functions from a Function Repository. Inputs to this algorithm are:

- a function repository,
- the group name of searched functions,
- arity of searched functions,
- argument information structures, and
- reference to a list of viable functions.

1. For each signature of the function repository that matches the group name, arity and argument infomation:
   (a) If a function with that signature is found in the function repository, it is inserted to the list of viable functions.
   (b) Otherwise, the function with the signature is created, inserted to the function repository, and inserted to the list of viable functions.

9.2 Derived Type Operation Repository

The following table shows functions contained by the derived type operation repository:

Signature	Condition	Function Symbol
`@constructor(P*)`	P is a pointer type	`DefaultCtor(P)`
`@constructor(P*, P)`	P is a pointer type	`CopyCtor(P)`
`@constructor(P*, const P&)`	P is a pointer type	`CopyCtor(P)`
`@constructor(P*, P&&)`	P is a pointer type	`MoveCtor(P)`
`@constructor(P*, void*)`	P is a pointer type	`ConvertingCtor(P, void*, @E)`
`@constructor(P*, @NP)`	P is a pointer type	`ConvertingCtor(P, @NP)`
`@constructor(P*, Q)`	(1)	`ConvertingCtor(P, Q, @E)`
`@constructor(C* D)`	(2)	`CopyCtor(C)`
`@constructor(C* D)`	(3)	`CopyCtor(C, @E)`
`@constructor(void**, void*)`		`CopyCtor(void*)`
`@constructor(void**, P)`	P is a pointer type	`ConvertingCtor(void*, P)`
`@constructor(R*, R)`	R is a reference type	`CopyCtor(R)`
`@constructor(E*, F)`	(4)	`CopyCtor(E)`
`@constructor(E*, F)`	(5)	`CopyCtor(E, @E)`
`@constructor(RR*, RR)`	RR is an rvalue reference type	`CopyCtor(RR)`
`operator=(P*, P)`	P is a pointer type	`CopyAssignment(P)`
`operator=(P*, const P&)`	P is a pointer type	`CopyAssignment(P)`
`operator=(P*, P&&)`	P is a pointer type	`MoveAssignment(P)`
`operator=(P*, @NP)`	P is a pointer type	`CopyAssignment(P, @NP)`

`operator=(C*, D)`	(2)	`CopyAssignment(C)`
`operator=(C*, D)`	(3)	`CopyAssignment(C, @E)`
`operator=(R*, R)`	R is a reference type	`CopyAssignment(R)`
`operator=(E*, F)`	(4)	`CopyAssignment(E)`
`operator=(E*, F)`	(5)	`CopyAssignment(E, @E)`
`operator=(RR*, RR)`	RR is an rvalue reference type	`CopyAssignment(RR)`
`operator==(P, P)`	P is a pointer type	`OpEqual(P)`
`operator==(P, @NP)`	P is a pointer type	`OpEqual(P)`
`operator==(@NP, P)`	P is a pointer type	`OpEqual(P)`
`operator==(C, D)`	(2)	`OpEqual(C)`
`operator==(C, D)`	(3)	`OpEqual(D)`
`operator<(P, P)`	P is a pointer type	`OpLess(P)`
`operator<(C, D)`	(2)	`OpLess(C)`
`operator+(P, I)`	(6)	`OpAddPtrInt(P)`
`operator+(I, P)`	(6)	`OpAddIntPtr(P)`
`operator-(P, P)`	P is a pointer type	`OpSubPtrPtr(P)`
`operator-(P, I)`	(6)	`OpSubPtrInt(P)`
`operator*(P)`	P is a pointer type	`OpDeref(P)`
`operator->(P)`	P is a pointer type	`OpArrow(P)`
`operator++(P)`	P is a pointer type	`OpIncPtr(P)`
`operator--(P)`	P is a pointer type	`OpDecPtr(P)`
`operator&(T)`		`OpAddrOf(@PT(T)*)`

@E = explicit conversion, i.e. requires a cast
@NP = null pointer type
@PT(T) = plain type of T
(1) P is a pointer type and Q is a pointer type
(2) C is a pointer type and D is a pointer type and base type of C is a class type and base type of D is a class type and D is derived from C
(3) C is a pointer type and D is a pointer type and base type of C is a class type and base type of D is a class type and C is derived from D
(4) E is a reference type and F is a reference type and base type of E is a class type and base type of F is a class type and F is derived from E
(5) E is a reference type and F is a reference type and base type of E is a class type and base type of F is a class type and E is derived from F
(6) P is a pointer type and I is an integer type

9.3 Enumerated Type Operation Repository

The following table shows functions contained by the enumerated type operation repository:

Signature	**Condition**	**Function Symbol**
`@constructor(E*)`	E is an enumerated type	`DefaultCtor(E)`
`@constructor(E*, E)`	E is an enumerated type	`CopyCtor(E)`
`@constructor(E*, const E&)`	E is an enumerated type	`CopyCtor(E)`
`@constructor(E*, E&&)`	E is an enumerated type	`MoveCtor(E)`

`@constructor(E*, I)`	(1)	`ConvertingCtor(E, @U(E), @E)`
`operator=(E*, E)`	E is an enumerated type	`CopyAssignment(E)`
`operator=(E*, const E&)`	E is an enumerated type	`CopyAssignment(E)`
`operator=(E*, E&&)`	E is an enumerated type	`MoveAssignment(E)`
`operator==(E, E)`	E is an enumerated type	`OpEqual(E)`
`operator<(E, E)`	E is an enumerated type	`OpLess(E)`

@E = explicit conversion, i.e. requires a cast
@U(E) = underlying type of E
(1) E is an enumerated type and I is an integer type

9.4 Array Type Operation Repository

The following table shows functions contained by the array type operation repository:

Signature	Condition	Function Symbol
`@constructor(A*)`	A is an array type	`ArrayTypeDefaultConstructor(A)`
`@constructor(A*, A)`	A is an array type	`ArrayTypeCopyConstructor(A)`
`@constructor(A*, const A&)`	A is an array type	`ArrayTypeCopyConstructor(A)`
`operator=(A*, A)`	A is an array type	`ArrayTypeCopyAssignment(A)`
`operator=(A*, const A&)`	A is an array type	`ArrayTypeCopyAssignment(A)`
`operator[](A*, I)`	(1)	`ArrayIndexing(A)`

(1) A is an array type and I is an integer type

9.5 Interface Type Operation Repository

The following table shows functions contained by the interface type operation repository:

Signature	Condition	Function Symbol
`@constructor(I*)`	I is an interface type	`InterfaceObjectDefaultCtor(I)`
`@constructor(I*, I)`	I is an interface type	`InterfaceObjectCopyCtor(I)`
`@constructor(I*, const I&)`	I is an interface type	`InterfaceObjectCopyCtor(I)`
`@constructor(I*, C*)`	(1)	`InterfaceObjectFromClassPtrCtor(I, C*)`
`operator=(I*, I)`	I is an interface type	`InterfaceObjectCopAssignment(I)`
`operator=(I*, const I&)`	I is an interface type	`InterfaceObjectCopAssignment(I)`
`operator==(I, I)`	I is an interface type	`InterfaceObjectOpEqual(I)`

(1) I is an interface type and C is a class type

9.6 Delegate Type Operation Repository

The following table shows functions contained by the delegate type operation repository:

Signature	Condition	Function Symbol
`@constructor(D*)`	D is a delegate type	`DefaultCtor(D)`
`@constructor(D*, D)`	D is a delegate type	`CopyCtor(D)`
`@constructor(D*, const D&)`	D is a delegate type	`CopyCtor(D)`
`@constructor(D*, D&&)`	D is a delegate type	`MoveCtor(D)`
`@constructor(D*, G)`	(1)	`DelegateFromFunCtor(D, @F(D, G))`
`operator=(D*, D)`	D is a delegate type	`CopyAssignment(D)`
`operator=(D*, const D&)`	D is a delegate type	`CopyAssignment(D)`
`operator=(D*, D&&)`	D is a delegate type	`MoveAssignment(D)`
`operator=(D*, G`	(1)	`DelegateFromFunAssignment(D, @F(D, G))`
`operator==(D, D)`	D is a delegate type	`OpEqual(D)`
`operator<(D, D)`	D is a delegate type	`OpLess(D)`

@F(D, G) = function symbol resolved from delegate type D and function group G
(1) D is a delegate type and G is a function group type

9.7 Class Delegate Type Operation Repository

The following table shows functions contained by the class delegate type operation repository:

Signature	Condition	Function Symbol
`@constructor(CD*)`	(1)	`ClassDelegateDefaultCtor(CD)`
`@constructor(CD*, const CD&)`	(1)	`ClassDelegateCopyCtor(CD)`
`@constructor(CD*, CD&&)`	(1)	`ClassDelegateMoveCtor(CD)`
`@constructor(CD*, C, G)`	(2)	`ClassDelegateFromFunCtor(CD, @F(CD, C, G))`
`operator=(CD*, const CD&)`	(1)	`ClassDelegateCopyAssignment(CD)`
`operator=(CD*, CD&&)`	(1)	`ClassDelegateMoveAssignment(CD)`
`operator=(CD*, C, G)`	(2)	`ClassDelegateFromFunAssignment(CD, @F(CD, C, G)`
`operator==(CD, CD)`	(1)	`ClassDelegateEqualOp(CD)`

@F(CD, C, G) = function symbol resolved from class delegate type CD, class type C and function group G
(1) CD is a class delegate type
(2) CD is a class delegate type, C is a class type and G is a function group type

9.8 Synthesized Class Function Repository

The following table shows functions contained by the synthesized class function repository:

Signature	Condition	Function Symbol
`@constructor(C*)`	C is a class type	`ClassDefaultCtor(C)`
`@constructor(C*, const C&)`	C is a class type	`ClassCopyCtor(C)`
`@constructor(C*, C&&)`	C is a class type	`ClassMoveCtor(C)`
`operator=(C*, const C&)`	C is a class type	`ClassCopyAssignment(C)`
`operator=(C*, C&&)`	C is a class type	`ClassMoveAssignment(C)`

`operator==(C, C)`	C is a class type	`ClassOpEqual(C)`

Chapter 10

Overload Resolution

We begin by describing the main algorithm for overload resolution, and then take a look at the details in the following sections.

10.1 Main Algorithm

The task of overload resolution is to select a single best matching function overload from a set of viable function overloads for a function call. Overload resolution proceeds as follows:

Algorithm 10.1.1. Overload Resolution. Inputs: The group name and arity of searched functions. A list of argument information structures (9.1). A list of <scope_kinds, scope> pairs, where scope_kinds is a combination of `this`, `base`, `parent` and `file` scope kinds, and scope is a container or file scope.

1. A *set of viable functions* is collected from the function repositories (chapter 9) and from the symbol table from the provided scopes. Each viable function has the same group name (see definition 4.1.1) and arity (see definition 4.1.2).

2. For each viable function:

   (a) If the viable function is an ordinary function, that is: not a function template, and the types of the arguments of the function call can be converted to the types of the viable function's parameters using algorithm 10.3, insert the viable function to a list of *matching functions*.

   (b) Otherwise, if the viable function is a function template: If the types of the arguments of the function call can be bound to the type parameters of the viable function template using algorithm 10.5.1, and the constraint of the function template is satisfied for the bound template argument types (algorithm **??**), insert the viable function template to a list of *matching functions*.

3. (a) If there are no matching functions, report error

   ```
   no matching functions found,
   or there are no acceptable conversion for all argument types
   ```

(b) Otherwise, if there is exactly one matching function, the overload resolution succeeds: If the matching function is a function template or member of a class template, it is instantiated with bound template arguments and the instance is returned; otherwise (the matching function is not a function template and not member of a class template) it is simply returned.

(c) Otherwise:

i. Sort the list of matching functions according to the function ordering rules (see section 10.4).

ii. If a single best matching function is found, the overload resolution succeeds: If the best matching function is a function template or member of a class template, it is instantiated with bound template arguments and the instance is returned; otherwise (the best matching function is not a function template and not member of a class template) it is simply returned.

iii. Otherwise, report ambiguous overload resolution error with references to ambiguous functions.

10.2 Examples

Consider the following example:

Example 10.2.1. Successful Overload Resolution.

```
1  public void foo(int x)
2  {
3      // ...
4  }
5
6  public void foo(long x)
7  {
8      // ...
9  }
10
11 void main()
12 {
13     foo(1);
14 }
```

There is two matching viable functions, `foo(int)` and `foo(long))`, for a function call `foo(1)` at line 13. Neither of them matches exactly, because `foo(sbyte)` would be an exact match. However, `foo(int)` is a better match than `foo(long)`, because the *conversion distance* of **sbyte** to **int**, i.e. 3, is less than the conversion distance of **sbyte** to **long**, i.e. 5. In this case the overload resolution is successful, and function `foo(int)` will be called.

Now, consider the following code:

Example 10.2.2. Unsuccessful Overload Resolution.

```
1 public void foo(int x, long y) {}
2 public void foo(long x, int y) {}
3
4 void main()
5 {
6     foo(1, 1);
7 }
```

In this case, the first overload, `foo(int, long)` is a better match for the first **sbyte** argument, but the second overload `foo(long, int)` is a better match for the second **sbyte** argument. However neither of them is better than the other according to the function ordering rules (section 10.4), so the overload resolution fails with an error:

```
Error: overload resolution for overload name 'foo(sbyte, sbyte)' failed:
call is ambiguous:
foo(int, long) or foo(long, int) (file 'C:/Temp/bind/bind.cm', line 6):
    foo(1, 1);
    ^^^^^^^^
see reference to file 'C:/Temp/bind/bind.cm', line 1:
public void foo(int x, long y)
            ^^^^^^^^^^^^^^^^^^
see reference to file 'C:/Temp/bind/bind.cm', line 2:
public void foo(long x, int y)
            ^^^^^^^^^^^^^^^^^^
```

10.3 Finding Conversions

The following algorithm is used for finding conversions from argument types to parameter types:

Algorithm 10.3.1. Find Conversions. Input to this algorithm are:

- List of parameters of a viable function.
- List of argument information structures (see section 9.1) for the function call to resolve.
- Conversion type `implicit` (default) or `explicit`.
- Let m be a reference to an initially empty list of argument matches associated with the viable function.

The algorithm returns **true** if conversions of argument types to parameter types exist, and **false** otherwise.

1. Let n be the number of parameters of the viable function and the number of argument information structures.
2. For $i = 1, \ldots, n$:

(a) Let a be i'th argument information structure. Let p the type of i'th parameter.

(b) If the type of argument in the argument information structure a is equal to the type of parameter p, append an `exactMatch` to the list of argument matches m.

(c) Otherwise:

i. If the parameter type p is a `nonconst` lvalue reference type and the category of argument in the argument information structore a is not `lvalue`, or the type of argument in the argument information structure a is `const` type, return **false**.

ii. Otherwise, if the parameter type p is an rvalue reference type, and the type of argument in the argument information structure a is not rvalue reference type and cannot bind to rvalue reference type according to the argument information structure a, return **false**.

(d) Otherwise, if the plain type of type of argument in the argument information structure a is equal to the plain type of p, append an `exactMatch` to the list of argument matches m with derivation counts of the argument type and the parameter type.

(e) Otherwise, if the plain type of the type of argument in the argument information structure a is an array type and plain type of p is pointer type, append a `conversion` match with distance 1 to the list of argument matches m with derivation counts of the argument type and the parameter type.

(f) Otherwise, if the base type of the type of argument in the argument information structure a is a class type and is derived from the base type of p, append a `conversion` match with the distance of argument class type to the parameter class type to the list of argument matches m with derivation counts of the argument type and the parameter type.

(g) Otherwise, if the conversion type is `explicit` and the base type of p is a class type and the the base type of the type of argument in the argument information structure a is a class type derived from p, append a `conversion` match with the distance of argument class type to the parameter class type to the list of argument matches m with derivation counts of the argument type and the parameter type.

(h) Otherwise, if there exists converting constructor or conversion function from plain parameter type to the plain argument type, or vice versa, let c be the converting constructor or conversion function, append a conversion argument match with the conversion distance of c to the list of argument matches m with derivation counts of the argument type and the parameter type.

(i) Otherwise, return **false**.

3. Return **true**.

10.4 Ordering of Matching Functions

The ordering of matching functions is based on lists of argument match structures associated with the matching functions and the properties of the matching functions themselves.

10.4.1 Argument Match Structures

An argument match structure contains:

- Conversion rank: `exactMatch`, `promotion`, `conversion`. When comparing these `exactMatch` is better than `promotion` and `promotion` is better than `conversion`. See section 4.2.1.5.
- Conversion distance. Shorter conversion distance is preferred over longer conversion distance.
- Parameter derivation counts. These are compared lexicographically: first the number of **const**s, then lvalue references, then rvalue references and finally pointers. The less derivations, the better.
- Argument derivation counts. Same comparison as above.

Argument match structures are compared lexicographically: first conversion ranks, then conversion distances, then parameter derivation counts and finally argument derivation counts.

10.4.2 Comparison Criteria Informally

When comparing two matching functions, we first compare their arguments.

- If the first function has more better matching arguments than the second one, we select the first, otherwise if the second function has more better matching arguments than the first one, we select the second.
- Then we compare total conversions made for these functions. If the first function requires fewer conversions than the second one, we select the first, otherwise if the second function requires fewer conversions than the first one, we select the second.
- Now the functions to compare have equal number of equally good (or equally bad) conversions, we prefer a function than is not a template and not instantiated from a template over a function template or a function that is instantiated from a function template.
- Now if they are both function templates or neither one is a function template, we have some special rules regarding array constructors and assignments.
- Now if they are both function templates:
  - We prefer a function template that has a constraint over a nonconstrained function template.
  - If both are constrained function templates, we use the *imply* relation to select the function template that has more strict constraint.

10.4.3 Comparison Algorithm

Ordering of matching functions is defined by comparing matching functions and their associated argument match structure lists pairwise:

Algorithm 10.4.1. Comparing Two Matching Functions. Returns **true** if the first matching function is a better match than the second matching function, and **false** otherwise.

1. Let $left$ be the first matching function and $right$ the second matching function.
2. Let la be the list of argument match structures for the parameters of $left$, and ra be the list of argument match structures for the parameters of $right$.
3. Let lb be the number of left better matching arguments and rb be the number right better matching arguments. Initially set lb and rb to zero.
4. For each argument match structure of la, say la_i and each corresponding argument match structure of ra, say ra_i:
   (a) If la_i is better argument match than ra_i, increment lb, else if ra_i is better argument match than la_i increment rb.
5. If $lb > rb$ return **true**, else if $rb > lb$ return **false**.
6. Otherwise, if the total number of conversions for $left$ is less than the total number of conversions for $right$, return **true**.
7. Otherwise, if the total number of conversions for $right$ is less than the total number of conversions for $left$, return **false**.
8. Otherwise, if $left$ is not a function template and $right$ is a function template, return **true**.
9. Otherwise, if $right$ is not a function template and $left$ is a function template, return **false**.
10. Otherwise, if $left$ is not a function template specialization and $right$ is a function template specialization, return **true**.
11. Otherwise, if $right$ is not a function template specialization and $left$ is a function template specialization, return **false**.
12. Otherwise, if $left$ is an array constructor and $right$ is not an array constructor, return **true**.
13. Otherwise, if $right$ is an array constructor and $left$ is not an array constructor, return **false**.
14. Otherwise, if $left$ is an array assignment and $right$ is not an array assignment, return **true**.
15. Otherwise, if $right$ is an array assignment and $left$ is not an array assignment, return **false**.
16. Otherwise, if $left$ has a constraint and $right$ does not have a constraint, return **true**.
17. Otherwise, if $right$ has a constraint and $left$ does not have a constraint, return **false**.
18. Otherwise, if both $left$ and $right$ have a constraint:

(a) Let lc be the bound constraint of the $left$ and rc be the bound constraint of the $right$.

(b) If lc imply rc, and not rc imply lc, return **true**.

(c) Otherwise, if rc imply lc and not lc imply rc, return **false**.

(d) Otherwise, return **false**.

19. Otherwise, return **false**.

10.5 Binding Types to Type Parameters

The next algorithms are used for deducing template arguments and binding them to template parameters. They are quite complicated as descibed using semiformal (bad) english. They can be used only directionally.

Algorithm 10.5.1. Deduce Template Arguments. Inputs: a container scope, list of template parameters, list of parameters, list of bound template arguments, reference to a list of template arguments a. The algorithm returns **true** if template arguments could be deduced, **false** otherwise.

1. Let n be the number of type parameters.
2. Resize the list of template arguments a be a list of n nulls.
3. Let m be the number of bound template arguments.
4. For $i = 1, \ldots, m$
   (a) Set template argument a_i to be bound template argument i.
5. Create a deduction scope, set its parent scope to container scope, and install template parameters to the deduction scope.
6. For each parameter p:
   (a) Let t be type expression for parameter p.
   (b) Resolve the type for t using the type resolver with deduction scope being the container scope. Let u be the type resolved.
   (c) If u is null, return **false**.
   (d) Use algorithm 10.5.2 to deduce a template argument and the type parameter for type u. Let b be the result of deduction.
   (e) If b is **false**, return **false**.
7. For $i = 1, \ldots, n$:
   (a) If template argument i is not bound (i.e. is null), return **false**.
8. Return **true**.

Algorithm 10.5.2. Deduce a Template Argument. Inputs: parameter type, argument type, list of template arguments. This algorithm returns **true** if template argument could be deduced, **false** otherwise.

1. If parameter type is equal to argument type, return **true**.
2. Let b be the result of binding of the argument type to the parameter type using algorithm 10.5.3 with template arguments.
3. If b = **true**, return **true**.
4. Otherwise, if an implicit conversion for argument type to parameter type exists, return **true**.
5. Otherwise, return **false**.

Algorithm 10.5.3. Binding Argument Type to Parameter Type. Inputs: parameter type, argument type, list of template arguments, reference to a bound type. This algorithm returns **true** if argument type could be bound to parameter type, **false** otherwise.

1. If the parameter type is a type parameter t:
   (a) Let i be the index of type parameter t.
   (b) If i'th template argument is null, set i'th template argument to argument type. Set bound type to argument type. Return **true**.
   (c) Otherwise, if i'th template argument is equal to the argument type. Set bound type to argument type. Return **true**.
   (d) Otherwise, return **false**.
2. Otherwise, if the base type of parameter type is class template specialization,
   (a) Let p be the base type of the argument type.
   (b) Let d be the derivation list of removed derivations (algorithm 10.5.4) for derivations of the argument type and derivations of the parameter type.
   (c) If the number of derivations in d is positive, set p to a derived type with derivations d and base type of the argument type.
   (d) If p is a class template specialization:
      i. if the primary class type of the base type of the parameter type is equal to the primary class type of p:
         A. Let n be the number of the type arguments of p.
         B. For $i = 1, \ldots, n$
            - If the i'th type argument of the base type of the parameter type is a type parameter:
              – Let k be the index of the i'th type argument of the base type of parameter type.
              – If k'th template argument is null, set k'th template argument to i'th type argument of p.

- Otherwise, if k'th template argument is not equal to the i'th type argument of p, return **false**.

C. Return **true**.

3. If the parameter type is a derived type symbol:
   (a) Let d be the derivation list of removed derivations (algorithm 10.5.4) for derivations of the argument type and derivations of the parameter type.
   (b) Let b be the base type of the argument type.
   (c) If the number of derivations in d is positive:
      i. Set b to a derived type with derivations d and base type of the argument type.
   (d) Let c be the result of binding of b to the base type of the parameter type using this algorithm.
   (e) If c is **true**, return **true**.
   (f) Otherwise, return **false**.
4. Otherwise, return **false**.

Algorithm 10.5.4. Remove Derivations. Inputs: list of target derivations, list of source derivations. This algorithm returns a list of derivations that survive when source derivations are removed from target derivations.

1. Let a derivation list r be empty.
2. Let a derivation list s be the list of source derivations.
3. Let n be the number of derivations in the list of target derivations.
4. For $i = 1, \ldots, n$:
   (a) Let t be the i'th target derivation.
   (b) Let m be the number of derivations in the source derivations.
   (c) Let $found$ be **false**.
   (d) For $j = 1, \ldots, m$:
      i. Let u be the j'th source derivation in s.
      ii. If $t = u$, set $found$ to **true**, and set the j'th source derivation in s to empty.
      iii. Otherwise, if t is `reference` and u is `rvalueref`, set $found$ to **true**, and set the j'th source derivation in s to empty.
   (e) If $found$ if **false**, add t to the derivation list r.
5. Return r.

10.6 Template Argument Deduction Example

Consider the following code:

Example 10.6.1. Template Argument Deduction Example.

```
1  using System;
2  using System.Collections;
3
4  void foo<T>(const T& x)
5  {
6      // ...
7  }
8
9  void bar(const List<int>& x)
10 {
11     foo(x);
12 }
```

We are now going to go through how type parameter T gets bound to type `List<int>` for the function call `foo(x)` using the previous algorithms.

1. 10.5.1 : 1.
   Let n be 1.
2. 10.5.1 : 2.
   $a = \{null\}$
3. 10.5.1 : 3.
   $m = 0$
4. 10.5.1 : 5.
   Create deduction scope: $\{T\} \rightarrow bar \rightarrow \ldots$
5. 10.5.1 : 6.
   For each parameter p:
6. 10.5.1 : 6. (a)
   p is parameter x. Let t be type expression node `const T&`
7. 10.5.1 : 6. (b)
   Resolve t. Let u be type symbol `const T&`
8. 10.5.1 : 6. (d)
   Use algorithm 10.5.2 to deduce template argument for type u:
9. 10.5.2: Parameter type = `const T&`. Argument type = `const List<int>&`. List of template arguments = $\{null\}$
10. 10.5.2 2.
    Use algorithm 10.5.3 to bind argument type to parameter type.

11. 10.5.3: Parameter type = `const T&`. Argument type = `const List<int>&`. List of template arguments = {*null*}

12. 10.5.3 3.
Parameter type is a derived type symbol `const T&`.

13. 10.5.3 3. (a)
Use algorithm 10.5.4 to remove derivations.

14. 10.5.4:
Target derivations = `const reference`. Source derivations = `const reference`.

15. 10.5.4 1.
Let derivation list r be {}.

16. 10.5.4 2.
List derivation list s be `const reference`.

17. 10.5.4 3.
Let n be 2.

18. 10.5.4 4.
For $i = 1, \ldots, 2$

19. 10.5.4 4. (a)
Let t be `const`

20. 10.5.4 4. (b)
Let m be 2.

21. 10.5.4 4. (c)
Let $found$ be **false**

22. 10.5.4 4. (d)
For $j = 1, \ldots, 2$

23. 10.5.4 4. (d) i. $j = 1$
Let u be `const`.

24. 10.5.4 4. (d) ii.
$t = u$ so set $found$ = **true**. Set `s[1]` = empty.

25. 10.5.4 4. (e)

26. 10.5.4 4. (a) $i = 2$
Let t be `reference`

27. 10.5.4 4. (b)
Let m be 2.

28. 10.5.4 4. (c)
Let $found$ be **false**

29. 10.5.4 4. (d)
    For $j = 1, \ldots, 2$

30. 10.5.4 4. (d) i. $j = 1$
    Let u be empty.

31. 10.5.4 4. (d) i. $j = 2$
    Let u be `reference`.

32. 10.5.4 4. (d) ii.
    $t = u$ so set $found =$ **true**. Set `s[2]` = empty.

33. 10.5.4 4. (e)

34. 10.5.4 5. return $r = \{\}$

35. 10.5.3 3. (a)
    Let d be $\{\}$.

36. 10.5.3 3. (b)
    Let b be the base type of `const List<int>&` i.e. $b =$ `List<int>`.

37. 10.5.3 3. (d)
    Call this algorithm 10.5.3 with parameter type T and argument type `List<int>`.

38. 10.5.3:
    Parameter type = `T`. Argument type = `List<int>`. List of template arguments = $\{null\}$

39. 10.5.3 1.
    Parameter type is type parameter T:

40. 10.5.3 1. (a)
    Let $i = 1$.

41. 10.5.3 1. (b)
    the first template argument is null so set the first the first template argument to `List<int>` and return **true**.

42. 10.5.3 3. (e)
    return **true**.

43. 10.5.1 : 6. (d)
    List of template arguments is now $a =$ `List<int>`.

44. 10.5.1 : 8.
    Return **true**.

Chapter 11

Concepts

In this chapter we will discuss the role of concepts in the overload resolution process, and then investigate the algorithms for checking and comparing constraints.

11.1 Concepts in Overload Resolution

Concepts are used in the overload resolution process (chapter 10) in two stages:

1. First concepts are used to *include* a constrained function template in the list of matching functions, provided that the deduced template arguments *satisfy* the constraint of the function template. After this stage we may have several function templates in the list of matching functions that have different constraints that are all satisfied.

2. The second stage is to *compare* bound constraints built from original constraints during constraint checking. The bound constraints are compared using the *imply* relation. This is done for selecting the best, i.e. the most strictly satisfying, constraint from the bound constraints. When comparing two constraints A and B: if $imply(A, B)$ is **true** and $imply(B, A)$ is **false**, then the function containing constraint A is a better match than the function containing B. Otherwise, if $imply(B, A)$ is **true** and $imply(A, B)$ is **false**, then the function containing B is a better match than the function containing A.

Example 11.1.1. Overload Resolution Using Concepts. The following listing shows some concepts that form iterator concept hierarchy in the System library:

```
1  public concept ForwardIterator<T> : InputIterator<T>
2  {
3      // ...
4  }
5
6  public concept BidirectionalIterator<T> : ForwardIterator<T>
7  {
8      // ...
9  }
10
11 public concept RandomAccessIterator<T> : BidirectionalIterator<T>
12 { /* ... */ }
```

We say that the `ForwardIterator` concept *refines* the `InputIterator` concept, the `BidirectionalIterator` concept refines the `ForwardIterator` concept, and the `RandomAccessIterator` concept refines the `BidirectionalIterator` concept. The refine relation is transitive, so that if a concept A refines a concept B and the concept B refines a concept C, then A refines C. When a concept A refines a concept B, the $imply(A, B)$ relation is **true** and $imply(B, A)$ relation is **false**.

Now consider following code:

```
1  public nothrow int Distance<I>(I first, I last) where I is
       ForwardIterator // [1]
2  {
3      int distance = 0;
4      while (first != last)
5      {
6          ++first;
7          ++distance;
8      }
9      return distance;
10 }
11
12 public nothrow inline int Distance<I>(I first, I last) where I is
       RandomAccessIterator // [2]
13 {
14     return last - first;
15 }
16
17 void main()
18 {
19     List<int> list;
20     int d1 = Distance(list.CBegin(), list.CEnd());
21     ForwardList<int> fwdList;
22     int d2 = Distance(fwdList.CBegin(), fwdList.CEnd());
23 }
```

The code example contains two implementations of `Distance` function, one for forward iterators, [1], and the other for random access iterators, [2].

The `List<int>.ConstIterator`, the iterator type that the `list.CBegin()` and `list.CEnd()` calls return, conforms to both RandomAccessIterator and ForwardIterator concepts, so for the `Distance` function call at line 20, both functions [1] and [2] are included in the list of matching functions. When comparing the constraints for functions [1] and [2], constraint for [2] wins because constraint of [2] imply constraint of [1], but not vice versa. Function [2], the random access iterator version, is called in this case.

The `ForwardList<int>.ConstIterator`, the iterator type that the `fwdList.CBegin()` and `fwdList.CEnd()` calls return, conforms only to ForwardIterator concept, so for the `Distance` function call at line 22, only function [1] is included in the list of matching functions. Function [1], the forward iterator version, is called in this case.

11.2 Checking and Binding Constraints

The following algorithm is used for checking and binding constraints:

Algorithm 11.2.1. Check Constraint. Inputs: a container scope, a constraint represented as an abstract syntax tree node, list of template parameters, list of template argument types, a reference to a bound constraint. The algorithm returns **true** if the constraint is satisfied, and **false** otherwise.

1. Create a scope for constraint checking and set its parent scope to the given container scope.
2. Let n be the number of template parameters. n is also the number of template argument types.
3. For $i = 1, \ldots, n$:
   (a) If $i = 1$, let $firstTypeArgument$ be i'th template argument type.
   (b) If $i = 2$, let $secondTypeArgument$ be i'th template argument type.
   (c) Let t be i'th template parameter. Let u be i'th template argument type.
   (d) Create a bound type parameter symbol b with name of t that maps name of t to u. Install b to the constraint checking scope.
4. Create an instance of constraint checker class that is an abstract syntax tree visitor.
5. Arguments to constraint checker constructor are: $firstTypeArgument$, $secondTypeArgument$, and a pointer to constraint checking scope.
6. Constraint checker has a stack of Boolean flags, a *constraintCheckStack*. It has also a stack of bound constraints. Finally it has a type, *resolvedType*, and a concept group symbol, *resolvedConceptGroup* that are used for resolving types and concept groups respectively.
7. Call the `Accept` member function of the constraint with constraint checker.
8. Pop the bound constraint from the stack of bound constraints and set it as the value of the bound constraint reference parameter.
9. Pop the result of the visitation, **true** or **false**, from the constraint check stack of the constraint checker, and return it.

The constraint checker overrides the following abstract syntax tree visitation points:

- `Visit(ConceptNode&)`: A `ConceptNode` contains:
  - group name of the concept.
  - type parameters of the concept.
  - refinement (optional): an abstract syntax tree node of type `ConceptId`.
  - constraints: a list of abstract syntax tree nodes derived from `ConstraintNode`.

  1. Call the `Accept` member function of the group name of the concept.

2. Let n be the number of type parameters of the concept.
3. Get concept symbol *concept* having n type parameters from *resolvedConceptGroup*.
4. If the `ConceptNode` has a refinement, do:
   (a) Call the `Accept` member function of the `ConceptNode` with this constraint checker visitor.
   (b) Pop the result of visiting the refinement, **true** or **false**, from the constraint check stack. Let r be the result.
   (c) Pop the *constraint* from the stack of bound constraints.
   (d) If r is **false**, push **false** to the constraint check stack, push *constraint* to the stack of bound constraints and return.
5. For each constraint in constraints contained by the `ConceptNode`:
   (a) Call the `Accept` member function of the constraint with this constraint checker visitor.
   (b) Pop the result of visiting the constraint, **true** or **false**, from the constraint check stack. Let c be the result.
   (c) Pop the *constraint* from the stack of bound constraints.
   (d) If c is **false**, push **false** to the constraint check stack, push *constraint* to the stack of bound constraints and return.
6. Push **true** to the constraint check stack.
7. Create a `BoundAtomicConstraint` with satisfied set to **true** and concept symbol set to *concept*, and push it to the stack of bound constraints.

- `Visit(ConceptIdNode& conceptIdNode)`: A `ConceptIdNode` contains:
  - a concept identifier.
  - list of type parameters.

1. Call the `Accept` member function of the identifier contained by the `ConceptIdNode` with this constraint checker visitor. If the *resolvedConceptGroup* is not null:
   (a) Let n be the number of number of type parameters containerd by the `ConceptIdNode`.
   (b) Get the concept symbol s with n type parameters from the *resolvedConceptGroup*.
   (c) List a be an empty list of type arguments.
   (d) For $i = 1, \ldots, n$
      i. Let t be the i'th type parameter contained by the `ConceptIdNode`.
      ii. Call the `Accept` member function of t with this constraint checker visitor.
      iii. If the *resolvedType* is not null:
         – Add *resolvedType* to the list of type arguments a.
   (e) Compute a 16-byte *conceptId* using algorithm 11.2.2 for the concept symbol s and list of type arguments a.
   (f) Lookup *conceptId* from the concept repository (section 11.2.1).
   (g) If found, push **true** to the constraint check stack.

(h) Otherwise, instantiate concept s with type arguments a using algorithm 11.2.3. Let c be the instantiated concept.

(i) If c is not null, add c to the concept repository with id *conceptId*, and push **true** to the constraint check stack.

(j) Otherwise, push **false** to the constraint check stack.

- `Visit(DisjunctiveConstraintNode& disjunctiveConstraintNode)`: `DisjunctiveConstraintNode` contains two constraint nodes, $left$ and $right$.

  1. Call `Accept` member function of $left$ with this constraint checking visitor.
  2. Pop the result of visiting the $left$ constraint, **true** or **false**, from the constraint check stack. Let c be the result.
  3. Pop bound constraint l from the stack of bound constraints.
  4. If c is **true**, push **true** to the constraint check stack, push bound constraint l to the stack of bound constraints and return.
  5. Otherwise, call `Accept` member function of $right$ with this constraint checking visitor.
  6. Pop the result of visiting the $right$ constraint, **true** or **false**, from the constraint check stack. Let s be the result.
  7. Pop bound constraint r from the stack of bound constraints.
  8. Push s to the constraint check stack.
  9. Create `BoundDisjunctiveConstraint` with l and r bound constraints and push it to the stack of bound constraints.

- `Visit(ConjunctiveConstraintNode& constructiveConstraintNode)`: `ConjunctiveConstraintNode` contains two constraint nodes, $left$ and $right$.

  1. Call `Accept` member function of $left$ with this constraint checking visitor.
  2. Pop bound constraint l from the stack of bound constraints.
  3. Pop the result of visiting the $left$ constraint, **true** or **false**, from the constraint check stack. Let c be the result.
  4. If c is **false**, push **false** to the constraint check stack, push l to the stack of bound constraints, and return.
  5. Otherwise, call `Accept` member function of $right$ with this constraint checking visitor.
  6. Pop bound constraint r from the stack of bound constraints.
  7. Pop the result of visiting the $right$ constraint, **true** or **false**, from the constraint check stack. Let s be the result.
  8. Push s to the constraint check stack.
  9. Create a `BoundConjunctiveConstraint` with l and r, and push it to the stack of bound constraints.

- `Visit(IdentifierNode& identifierNode)`:

1. Set *resolvedType* to null and *resolvedConceptGroup* to null.
2. Lookup a string contained by the identifierNode from the container scope. Let s be the symbol found.
3. If s is not null:
   (a) If s is a type symbol, set *resolvedType* to s and return.
   (b) Otherwise, if s is a bound type parameter symbol, set *resolvedType* to a mapped type symbol of the bound type parameter symbol and return.
   (c) Otherwise, if s is a typedef symbol, set *resolvedType* to the type contained by the typedef symbol and return.
   (d) Otherwise, if s is a concept group symbol, set *resolvedConceptGroup* to s and return.
   (e) Otherwise, if s is a namespace symbol, create a namespace type symbol containing namespace s and set $resolvedType$ to that namespace type symbol and return.
4. Otherwise, report error.

- `EndVisit(DotNode& dotNode)`:

1. If *resolvedType* is null, report error and return.
2. Let $typeContainerScope$ be the container scope of $resolvedType$.
3. if $resolvedType$ is a namespace type symbol, set $typeContainerScope$ to the container scope of the namespace contained by $resolvedType$.
4. Lookup a symbol with name contained by the dotNode from the $typeContainerScope$. Let s be the symbol found.
5. If s is not null:
   (a) If s is a bound type parameter symbol, set *resolvedType* to a mapped type symbol of the bound type parameter symbol and return.
   (b) Otherwise, if s is a typedef symbol, set *resolvedType* to the type contained by the typedef symbol and return.
   (c) Otherwise, if s is a concept group symbol, set *resolvedConceptGroup* to s and return.
   (d) Otherwise, if s is a namespace symbol, create a namespace type symbol containing namespace s and set $resolvedType$ to that namespace type symbol and return.
6. Otherwise, report error.

- `Visit(BoolNode& boolNode)`:
  Set *resolvedType* to `BoolTypeSymbol`.

- `Visit(SByteNode& sbyteNode)`:
  Set *resolvedType* to `SByteTypeSymbol`.

- `Visit(ByteNode& byteNode)`:
  Set *resolvedType* to `ByteTypeSymbol`.

- `Visit(ShortNode& shortNode)`:
  Set *resolvedType* to `ShortTypeSymbol`.
- `Visit(UShortNode& ushortNode)`:
  Set *resolvedType* to `UShortTypeSymbol`.
- `Visit(IntNode& intNode)`:
  Set *resolvedType* to `IntTypeSymbol`.
- `Visit(UIntNode& uintNode)`:
  Set *resolvedType* to `UIntTypeSymbol`.
- `Visit(LongNode& longNode)`:
  Set *resolvedType* to `LongTypeSymbol`.
- `Visit(ULongNode& ulongNode)`:
  Set *resolvedType* to `ULongTypeSymbol`.
- `Visit(FloatNode& floatNode)`:
  Set *resolvedType* to `FloatTypeSymbol`.
- `Visit(DoubleNode& doubleNode)`:
  Set *resolvedType* to `DoubleTypeSymbol`.
- `Visit(CharNode& charNode)`:
  Set *resolvedType* to `CharTypeSymbol`.
- `Visit(WCharNode& wcharNode)`:
  Set *resolvedType* to `WCharTypeSymbol`.
- `Visit(UCharNode& ucharNode)`:
  Set *resolvedType* to `UCharTypeSymbol`.
- `Visit(VoidNode& voidNode)`:
  Set *resolvedType* to `VoidTypeSymbol`.
- `Visit(DerivedTypeExprNode& derivedTypeExprNode`:
  Resolve type contained by derivedTypeExprNode using type resolver. Set *resolvedType* to the type resolved.
- `Visit(IsConstraintNode& isConstrainedNode`:
  `IsConstraintNode` contains:
  - type expression
  - name of a concept or a type

  1. Call `Accept` member function of the type expression of the isConstrainedNode with this constraint checking visitor.
  2. Let $leftType$ be the result of visitation, that is: *resolvedType*.
  3. Call `Accept` member function of the name of a concept or type with this constraint checking visitor.

4. If *resolvedType* is not null:
   (a) Make plain type for $leftType$ using algorithm 5.4.8. Let $leftPlainType$ be the result.
   (b) Make plain type for *resolvedType* using algorithm 5.4.8. Let $rightPlainType$ be the result.
   (c) If $leftPlainType$ equals $rightPlainType$, push **true** to the constraint check stack, and push a `BoundAtomicConstraint` with value **true** to the stack of bound constraints.
   (d) Otherwise, push **false** to the constraint check stack, and push a `BoundAtomicConstraint` with value **false** to the stack of bound constraints.
5. Otherwise, if *resolvedConceptGroup* is not null:
   (a) Get concept symbol s with one type parameter from *resolvedConceptGroup*.
   (b) Let a be a list of type arguments.
   (c) Add $leftType$ to a.
   (d) Compute a 16-byte $conceptId$ using algorithm 11.2.2 for the concept symbol s and list of type arguments a.
   (e) Lookup $conceptId$ from the concept repository (section 11.2.1).
   (f) If found, push **true** to the constraint check stack, and push a bound constraint cloned from the bound constraint contained by the instantiated concept to the stack of bound constraints. and return.
   (g) Otherwise, instantiate concept s with type arguments a using algorithm 11.2.3. Let c be the instantiated concept.
   (h) If c is not null, add c to the concept repository with id $conceptId$, push **true** to the constraint check stack, push a bound constraint corresponding to c to the stack of bound constraints, and return.
   (i) Otherwise, push **false** to the constraint check stack, push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(MultiParamConstraintNode& multiParamConstraintNode)`: `MultiParamConstraintNode` contains:
  - Identifier node that should contain a name of a concept group.
  - List of type expressions.

1. Call `Accept` member function of the identifier node contained by multiParamConstraintNode.
2. If *resolvedConceptGroup* is not null:
   (a) Let n be the number of type expressions contained by multiParamConstraintNode.
   (b) Get concept symbol s with n type parameters from the *resolvedConceptGroup*.
   (c) Let a be a list of type arguments.
   (d) For $i = 1, \ldots, n$:
      i. Let t be a i'th type expression of the multiParamConstraintNode.
      ii. Call the `Accept` member function of t with this constraint checker visitor.

iii. If *resolvedType* is not null, add *resolvedType* to *a*.

iv. Otherwise report error.

(e) Compute a 16-byte *conceptId* using algorithm 11.2.2 for the concept symbol *s* and list of type arguments *a*.

(f) Lookup *conceptId* from the concept repository (section 11.2.1).

(g) If found, push **true** to the constraint check stack, push a bound constraint cloned from the bound constraint contained by the instantiated concept, and return.

(h) Otherwise, instantiate concept *s* with type arguments *a* using algorithm 11.2.3. Let *c* be the instantiated concept.

(i) If *c* is not null, add *c* to the concept repository with id *conceptId*, and push **true** to the constraint check stack, and push bound constraint corresponding to *c* to the stack of bound constraints.

(j) Otherwise, push **false** to the constraint check stack and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

3. Otherwise report error.

- `Visit(TypenameConstraintNode& typenameConstraintNode)`:
  `TypenameConstraintNode` contains an abstract syntax tree node *typeId* that represents a type associated with another type.

  1. Call the `Accept` member function of *typeId* with this constraint checker visitor.
  2. If *resolvedType* is not null, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints.
  3. Otherwise, push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(ConstructorConstraintNode& constructorConstraintNode)`:

  1. Resolve parameter types represented by parameter nodes in the constructorConstraintNode using type resolver.
  2. Then lookup a constructor within a class type represented by $firstTypeArgument$ having those parameter types using overload resolution.
  3. If a constructor is found, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints.
  4. Otherwise, push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(DestructorConstraintNode& destructorConstraintNode)`:
  Push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints.

- `Visit(MemberFunctionConstraintNode& memberFunctionConstraintNode)`:

  1. Call the `Accept` member function of the type parameter identifier of the memberFunctionConstraintNode.

2. Let the first parameter type be pointer to *resolvedType*.
3. Let the group name of the member function be group id of the memberFunctionConstraintNode.
4. Resolve other parameter types represented by parameter nodes in the memberFunctionConstraintNode using type resolver.
5. Then lookup a member function having the group name and these parameter types using overload resolution.
6. If a member function is found, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints.
7. Otherwise, push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(FunctionConstraintNode& functionConstraintNode)`:

1. Let the group name of the function be group id of the functionConstraintNode.
2. Resolve parameter types represented by parameter nodes in the functionConstraintNode using type resolver.
3. Then lookup a function having the group name and these parameter types using overload resolution.
4. If a function is found, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints.
5. Otherwise, push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(SameConstraintNode& sameConstraintNode)`:
  If types $firstArgumentType$ and *secondArgumentType* are same, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints. Otherwise push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(DerivedConstraintNode& derivedConstraintNode)`:
  If type $firstArgumentType$ is derived from the *secondArgumentType*, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints. Otherwise push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(ConvertibleConstraintNode& convertibleConstraintNode)`:
  If type $firstArgumentType$ is implicitly convertible to the *secondArgumentType*, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints. Otherwise push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(ExplicitlyConvertibleConstraintNode& explicitlyConvertibleConstraintNode)`:
  If type $firstArgumentType$ is explicitly convertible to the *secondArgumentType*, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints. Otherwise push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(CommonConstraintNode& commonConstraintNode)`:
  If types $firstArgumentType$ and $secondArgumentType$ are same, let $commonType$ be $firstArgumentType$, otherwise, if $firstArgumentType$ is convertible to the $secondArgumentType$, let $commonType$ be $secondArgumentType$, otherwise, if $secondArgumentType$ is convertible to the $firstArgumentType$, let $commonType$ be $firstArgumentType$, otherwise, let $commonType$ be null. If $commonType$ is not null, install $commonType$ to the container scope, and push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints. Otherwise push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

- `Visit(NonReferenceTypeConstraintNode& nonReferenceTypeConstraintNode)`:
  If $firstArgumentType$ is not lvalue reference type and not rvalue reference type, push **true** to the constraint check stack, and push `BoundAtomicConstraint` with value **true** to the stack of bound constraints. Otherwise push **false** to the constraint check stack, and push `BoundAtomicConstraint` with value **false** to the stack of bound constraints.

11.2.1 Concept Repository

Each concept symbol contains a unique 16-byte type identifier computed using Mersenne Twister pseudorandom number generator (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html). An instantiated concept has also a 16-byte identifier that is formed by the type identifier of the concept symbol **xor**ed with the rotated type identifiers of the types for which the concept is instantiated.

The concept repository keeps mapping from identifiers computed for instantiated concepts to the instantiated concepts themselves. The instantiated concepts are cached per compilation unit basis. The following algorithm computes these identifiers:

Algorithm 11.2.2. Computing 16-byte Identifier for an Instantiated Concept. Inputs: A concept symbol, list of type arguments. The algorithm returns a 16-byte identifier for a concept instantiated with given type arguments.

1. Let id be the type identifier of the concept symbol.
2. Let n be the number of type arguments.
3. For $i = 0, \ldots, n - 1$:
   (a) Let a be the type identifier for i'th type argument.
   (b) Let r be a rotated by i byte positions right.
   (c) Assign id **xor** r to id.
4. Return id.

11.2.2 Instantiating a Concept

The following algorithm is used to instantiate a concept with type arguments:

Algorithm 11.2.3. Instantiate a Concept with Type Arguments. Inputs: a container scope, a concept symbol s, list of type arguments a, a reference to a bound constraint. The algorithm returns an instantiated concept symbol, or null if instantiation did not succeed.

1. Let n be the number of type parameters of concept symbol s.
2. Create an instantiation scope and set its parent scope to container scope.
3. For $i = 1, \ldots, n$:
   (a) Let p be the i'th type parameter of concept symbol s.
   (b) Let t be the i'th type argument of a.
   (c) If $i = 1$ let $firstTypeArgument$ be t.
   (d) If $i = 2$ let $secondTypeArgument$ be t.
   (e) Create a bound type parameter symbol with name of p mapped to type t and install it to the instantiation scope.
4. Create a constraint checker with $firstTypeArgument$, $secondTypeArgument$ and instantiation scope.
5. Let c be the concept node corresponding to the concept symbol.
6. Call the `Accept` member function of c with the constraint checker visitor.
7. Let r be the result of visitation, **true** or **false**.
8. Get bound constraint b from the constraint checker.
9. If r is **true**, create an instantiated concept symbol for s with type arguments a and bound constraint b, and return it.
10. Otherwise, return null.

11.3 Comparing Constraints

The following listing shows hierarchy of bound constraints:

```
BoundNode
    BoundConstraint
        BoundAtomicConstraint
        BoundBinaryConstraint
            BoundDisjunctiveConstraint
            BoundConjunctiveConstraint
```

A `BoundAtomicConstraint` contains a Boolean flag that is equal to the evaluated result of the corresponding unbounded constraint. It can also contain a concept symbol if it is result of evaluation of a concept constraint. A `BoundBinaryConstraint` contains two child constraints, $left$ and $right$.

The following algorithm is used to compare bound constraints to find out which of them is the most stricly satisfying constraint:

Algorithm 11.3.1. Imply. Inputs: two bound constraints A and B. The algorithm returns **true** if $imply(A, B)$ is **true** and **false** otherwise.

1. If A is `BoundAtomicConstraint`:
   (a) If B is `BoundBinaryConstraint`:
      i. Let LB be the left constraint of B.
      ii. Let RB be the right constraint of B.
      iii. Let L be $imply(A, LB)$.
      iv. Let R be $imply(A, RB)$.
      v. If B is `BoundConjunctiveConstraint`, return L **and** R.
      vi. Otherwise, B is `BoundDisjunctiveConstraint`, return L **or** R.
   (b) Otherwise, B is `BoundAtomicConstraint`:
      i. If A is satisfied and B is not satisfied, return **false**.
      ii. Otherwise, if A contains a concept symbol and B does not contain a concept symbol, return **false**.
      iii. Otherwise, if A does not contain a concept symbol and B contains a concept symbol, return **true**.
      iv. Otherwise, if both A and B contain a concept symbol:
         A. If concept symbol contained by A is equal to the concept symbol contained by B, return **true**.
         B. Otherwise, let $refinedConcept$ be the refined concept of the concept symbol contained by A.
         C. While $refinedConcept$ is not null:
            - If $refinedConcept$ is equal to the concept symbol contained by B, return **true**.
            - Otherwise, set $refinedConcept$ to the refined concept of the $refinedConcept$.
         D. Return **false**.
      v. Otherwise, return **true**.
2. Otherwise, if A is `BoundDisjunctiveConstraint`,
   (a) Let AL be the left constraint of A.
   (b) Let AR be the right constraint of A.
   (c) If B is `BoundBinaryConstraint`,
      i. Let BL be the left constraint of B.
      ii. Let BR be the right constraint of B.

iii. Let LL be $imply(AL, BL)$.
iv. Let LR be $imply(AL, BR)$.
v. Let RL be $imply(AR, BL)$.
vi. Let RR be $imply(AR, BR)$.
vii. Let $LLorLR$ be LL **or** LR.
viii. Let $RLorRR$ be RL **or** RR.
ix. If B is `BoundConjunctiveConstraint`, return $LLorLR$ **and** $RLorRR$.
x. Otherwise, B is `BoundDisjunctiveConstraint`, return $LLorLR$ **or** $RLorRR$.

(d) Otherwise, B is `BoundAtomicConstraint`:
i. Let LB be $imply(AL, B)$.
ii. Let RB be $imply(AR, B)$.
iii. Return LB **and** RB.

3. Otherwise, if A is `BoundConjunctiveConstraint`,

(a) Let AL be the left constraint of A.
(b) Let AR be the right constraint of A.
(c) If B is `BoundBinaryConstraint`,
i. Let BL be the left constraint of B.
ii. Let BR be the right constraint of B.
iii. Let LL be $imply(AL, BL)$.
iv. Let RL be $imply(AR, BL)$.
v. Let LR be $imply(AL, BR)$.
vi. Let RR be $imply(AR, BR)$.
vii. Let $LorRL$ be LL **or** RL.
viii. Let $LorRR$ be LR **or** RR.
ix. If B is `BoundConjunctiveConstraint`, return $LorRL$ **and** $LorRR$.
x. Otherwise, B is `BoundDisjunctiveConstraint`, return $LorRL$ **or** $LorRR$.

(d) Otherwise, B is `BoundAtomicConstraint`:
i. Let LB be $imply(AL, B)$.
ii. Let RB be $imply(AR, B)$.
iii. Return LB **or** RB.

Chapter 12

Binding Expressions

You may want to recall what is said about the *bound tree* representation in section 1.4. The component that creates bound expression nodes from abstract syntax tree expression nodes is called the *expression binder*. The expression binder has a stack of bound expressions that contains the intermediate and final bound expression nodes. We begin by investigating the bound expression node hierarchy.

12.1 Bound Expression Node Hierarchy

Here's the bound expression node hierarchy:

```
BoundNode
    BoundExpression
        BoundLiteral
        BoundStringLiteral
        BoundConstant
            BoundExceptionTableConstant
            BoundClassHierarchyTableConstant
        BoundEnumConstant
        BoundLocalVariable
            BoundExceptionCodeVariable
        BoundParameter
            BoundExceptionCodeParameter
        BoundReturnValue
        BoundMemberVariable
        BoundFunctionId
        BoundTypeExpression
        BoundNamespaceExpression
        BoundUnaryOp
        BoundBinaryOp
        BoundFunctionCall
        BoundDelegateCall
        BoundClassDelegateCall
        BoundConversion
```

```
BoundCast
BoundIsExpression
BoundAsExpression
BoundSizeOfExpression
BoundDynamicTypeNameExpression
BoundBooleanBinaryExpression
    BoundDisjunction
    BoundConjunction
```

Each kind of bound expression has a *type* that is represented by a symbol class derived from the `TypeSymbol` class (see section 7.1). Many expressions are represented as instances of `BoundUnaryOp` and `BoundBinaryOp` classes. Therefore we begin by describing the algorithms that bind unary and binary operators.

12.2 Binding Unary and Binary Operators

When binding unary and binary operators the expression binder first visits the abstract syntax tree nodes that represents operands of the unary or binary operator, creates bound nodes for the operands and pushes them to the bound expression stack. Then the abstract syntax tree node for the operator is visited. At this point the expression binder pops the operands from the bound expression stack and resolves the operator function symbol using overload resolution. It then creates `BoundUnaryOp` or `BoundBinaryOp` node that is bound to the operator function symbol and pushes it to the bound expression stack.

Algorithm 12.2.1. Bind Unary Operator. Inputs: An abstract syntax tree node that represents the unary operator. The group name of the unary operator function. The algorithm first tries to bind the unary operator to a member function symbol. If that does not succeed, it binds the unary operator to a nonmember function symbol. The group names for the unary operator functions bound using this algorithm are:

1. `operator!`
2. `operator++`
3. `operator--`
4. `operator+`
5. `operator-`
6. `operator~`
7. `operator&`
8. `operator*`
9. `operator->`

The steps for binding a unary operator are:

1. Pop the *operand* from the bound expression stack.

2. Let t be the plain type for the type of the *operand* (algorithm 5.4.8).
3. Let *conversions* be an empty list of function symbols.
4. Let *memFunArgs* be a list of argument information structures (9.1).
5. Let *memFunLookups* be a list of scope lookups.
6. Add <class scope of t, `this` | `base`> to *memFunLookups*.
7. Let p be of type pointer to t. Add p to the *memFunArgs*.
8. Call overload resolution algorithm 10.1.1 with the group name of the unary operator function, *memFunArgs*, *memFunLookups* and *conversions*.
9. Set fun to a function symbol resolved.
10. If resolution did not succeed,
    (a) Let $freeFunArgs$ be a list of argument information structures (9.1).
    (b) Add t and argument category of t to $freeFunArgs$.
    (c) Let $freeFunLookups$ be a list of scope lookups.
    (d) Add <container scope, `this` | `base` | `parent`> to $freeFunLookups$.
    (e) Add <namespace scope of t, `this` | `base` | `parent`> to $freeFunLookups$.
    (f) Add <file scope, `file`> to $freeFunLookups$.
    (g) Call overload resolution algorithm 10.1.1 with the group name of the unary operator function, $freeFunArgs$, $freeFunLookups$ and *conversions*.
    (h) Set fun to a function symbol resolved.
11. If *conversions*[0] is not null, replace *operand* with a `BoundConversion` node containing a conversion function *conversions*[0] and *operand*.
12. Create a `BoundUnaryOp` *op* with a function symbol fun and operand *operand*.
13. Set the type of *op* to the return type of fun.
14. Push *op* to the bound expression stack.

Algorithm 12.2.2. Bind Binary Operator. Inputs: An abstract syntax tree node that represents the binary operator. The group name of the binary operator function. The algorithm first tries to bind the binary operator to a member function symbol. If that does not succeed, it binds the binary operator to a nonmember function symbol. The group names for the binary operator functions bound using this algorithm are:

1. `operator|`
2. `operator^`
3. `operator&`
4. `operator==`

5. `operator<`
6. `operator<<`
7. `operator>>`
8. `operator+`
9. `operator-`
10. `operator*`
11. `operator/`
12. `operator%`
13. `operator[]`

The steps for binding a binary operator are:

1. Pop the *rightOperand* from the bound expression stack.
2. Pop the $leftOperand$ from the bound expression stack.
3. Let t be the plain type for the type of the $leftOperand$ (algorithm 5.4.8).
4. Let *conversions* be an empty list of function symbols.
5. Let $memFunArgs$ be a list of argument information structures (9.1).
6. Let $memFunLookups$ be a list of scope lookups.
7. Add <class scope of t, `this` | `base`> to $memFunLookups$.
8. Let p be of type pointer to t. Add p to the $memFunArgs$.
9. Add the type of the *rightOperand* and the argument category of the *rightOperand* to $memFunArgs$.
10. Call overload resolution algorithm 10.1.1 with the group name of the binary operator function, $memFunArgs$, $memFunLookups$ and *conversions*.
11. Set fun to a function symbol resolved.
12. If resolution did not succeed,
    (a) Let $freeFunArgs$ be a list of argument information structures (9.1).
    (b) Add t and argument category of t to $freeFunArgs$.
    (c) Add the type of the *rightOperand* and the argument category of the *rightOperand* to $freeFunArgs$.
    (d) Let $freeFunLookups$ be a list of scope lookups.
    (e) Add <container scope, `this` | `base` | `parent`> to $freeFunLookups$.
    (f) Add <namespace scope of t, `this` | `base` | `parent`> to $freeFunLookups$.

(g) Add <namespace scope of type of *rightOperand*, `this` | `base` | `parent`> to *freeFunLookups*.

(h) Add <file scope, `file`> to *freeFunLookups*.

(i) Call overload resolution algorithm 10.1.1 with the group name of the binary operator function, *freeFunArgs*, *freeFunLookups* and *conversions*.

(j) Set *fun* to a function symbol resolved.

13. If *conversions*[0] is not null, replace *leftOperand* with a `BoundConversion` containing a conversion function *conversions*[0] and *leftOperand*.

14. If *conversions*[1] is not null, replace *leftOperand* with a `BoundConversion` containing a conversion function *conversions*[1] and *rightOperand*.

15. Create a `BoundBinaryOp` *op* with a function symbol *fun* and operands *leftOperand* and *rightOperand*.

16. Set the type of *op* to the return type of *fun*.

17. Push *op* to the bound expression stack.

12.3 Expression Binder

Algorithm 12.3.1. Binding an Expression. The expression binder is an abstract syntax tree visitor. It overrides the following visitation points:

- `EndVisit(BitOrNode& bitOrNode)`:
  Calls algorithm 12.2.2 with bitOrNode and group name `"operator|"`.

- `EndVisit(BitXorNode& bitXorNode)`:
  Calls algorithm 12.2.2 with bitXorNode and group name `"operator^"`.

- `EndVisit(BitAndNode& bitAndNode)`:
  Calls algorithm 12.2.2 with bitAndNode and group name `"operator&"`.

- `EndVisit(EqualNode& equalNode)`:
  Calls algorithm 12.2.2 with equalNode and group name `"operator=="`.

- `EndVisit(NotEqualNode& notEqualNode)`:
  Note: `a != b <=> !(a == b)`
  Calls algorithm 12.2.2 with notEqualNode and group name `"operator=="`.
  Calls algorithm 12.2.1 with notEqualNode and group name `"operator!"`.

- `EndVisit(LessNode& lessNode)`:
  Calls algorithm 12.2.2 with notEqualNode and group name `"operator<"`.

- `EndVisit(GreaterNode& greaterNode)`:
  Note: `a > b <=> b < a`
  Exchange the operands in the bound expression stack and then bind `operator<`:

  1. Pop *rightOperand* from the bound expression stack.

2. Pop *leftOperand* from the bound expression stack.
3. Push *rightOperand* to the bound expression stack.
4. Push *leftOperand* to the bound expression stack.
5. Call algorithm 12.2.2 with greaterNode and group name `"operator<"`.

- `EndVisit(LessOrEqualNode& lessOrEqualNode)`:
  Note: `a <= b <=> !(b < a)`
  Exchange the operands in the bound expression stack, and then bind `operator<` and `operator!`:

  1. Pop *rightOperand* from the bound expression stack.
  2. Pop *leftOperand* from the bound expression stack.
  3. Push *rightOperand* to the bound expression stack.
  4. Push *leftOperand* to the bound expression stack.
  5. Call algorithm 12.2.2 with lessOrEqualNode and group name `"operator<"`.
  6. Call algorithm 12.2.1 with lessOrEqualNode and group name `"operator!"`.

- `EndVisit(GreaterOrEqualNode& greaterOrEqualNode)`:
  Note: `a >= b <=> !(a < b)`
  Calls algorithm 12.2.2 with greaterOrEqualNode and group name `"operator<"`.
  Calls algorithm 12.2.1 with greaterOrEqualNode and group name `"operator!"`.

- `EndVisit(ShiftLeftNode& shiftLeftNode)`:
  Calls algorithm 12.2.2 with shiftLeftNode and group name `"operator<<"`.

- `EndVisit(ShiftRightNode& shiftRightNode)`:
  Calls algorithm 12.2.2 with shiftRightNode and group name `"operator>>"`.

- `EndVisit(AddNode& addNode)`:
  Calls algorithm 12.2.2 with addNode and group name `"operator+"`.

- `EndVisit(SubNode& subNode)`:
  Calls algorithm 12.2.2 with subNode and group name `"operator-"`.

- `EndVisit(MulNode& mulNode)`:
  Calls algorithm 12.2.2 with mulNode and group name `"operator*"`.

- `EndVisit(DivNode& divNode)`:
  Calls algorithm 12.2.2 with divNode and group name `"operator/"`.

- `EndVisit(RemNode& remNode)`:
  Calls algorithm 12.2.2 with remNode and group name `"operator%"`.

- `EndVisit(PrefixIncNode& prefixIncNode)`:
  Calls algorithm 12.2.1 with prefixIncNode and group name `"operator++"`.

- `EndVisit(PrefixDecNode& prefixDecNode)`:
  Calls algorithm 12.2.1 with prefixDecNode and group name `"operator--"`.

- `EndVisit(UnaryPlusNode& unaryPlusNode)`:
  Calls algorithm 12.2.1 with unaryPlsuNode and group name `"operator+"`.
- `EndVisit(UnaryMinusNode& unaryMinusNode)`:
  Calls algorithm 12.2.1 with unaryMinusNode and group name `"operator-"`.
- `EndVisit(NotNode& notNode)`:
  Calls algorithm 12.2.1 with notNode and group name `"operator!"`.
- `EndVisit(ComplementNode& complementNode)`:
  Calls algorithm 12.2.1 with complementNode and group name `"operator~"`.
- `Visit(AddrOfNode& addrOfNode)`:
  Calls algorithm 12.2.1 with addrOfNode and group name `"operator&"`.
- `Visit(DerefNode& derefNode)`:
  Calls algorithm 12.2.1 with derefNode and group name `"operator*"`.

Chapter 13

Binding Statements

Chapter 14

Template Repositories

Chapter 15

Binding Compile Units

Chapter 16

Emitter

Bibliography

[1] Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman: Compilers: Principles, Techniques, & Tools. Second Edition. Addison-Wesley, 2007.

[2] Hopcroft, J. E., R. Motwani, and J. D. Ullman: Introduction to Automata Theory, Languages, and Computation. Second Edition. Addison-Wesley, 2001.

[3] Joel de Guzman: Spirit Parsing Libraries, *http://boost-spirit.com/home/*

[4] LLVM team: LLVM Language Reference Manual, *http://llvm.org/docs/LangRef.html*