Implementation of the Cmajor Compiler

Seppo Laakko

March 28, 2016

Contents

Chapter 1

Introduction

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 1.0.1. Consider the following source code fragment:

```
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 `;`.

[1] type for example

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')
```

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.

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
```

[2]this is LLVM intermediate code [4]

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)
```

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.

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

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

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

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`.

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 Lexical Analysis in Cmajor

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
$r_1 \vert r_2$	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:

```
grammar CompileUnitGrammar
{
    // ...
    skip spaces_and_comments;
    // ...
}
```

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

```
spaces_and_comments
    ::= (space | comment)+
    ;

comment
    ::= line_comment | block_comment
    ;

line_comment
    ::= "//" [^\r\n]* newline
    ;

newline
    ::= "\r\n" | "\n" | "\r"
    ;

block_comment
    ::= "/*" (StringLiteral | CharLiteral | (anychar - "*/"))* "*/"
    ;
```

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:

```
BooleanLiteral
    ::= keyword("true")
    |   keyword("false")
    ;
```

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 for 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 for 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 $\overset{*}{\Rightarrow}$ symbol. For example, to indicate that there is a derivation of string 0110 from variable P using some number of steps, is denoted $P \overset{*}{\Rightarrow} 0110$.

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.

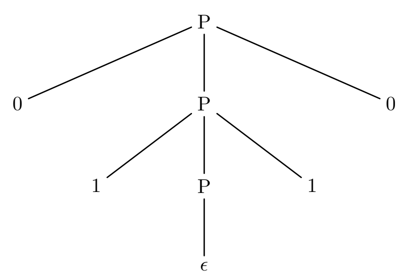
3. If an interior node is labeled A, and its children are labeled

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

respectively, from the left, then $A \rightarrow X_1 X_2 \cdots X_k$ is a production of the grammar G.

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

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



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

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

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.

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

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 *.

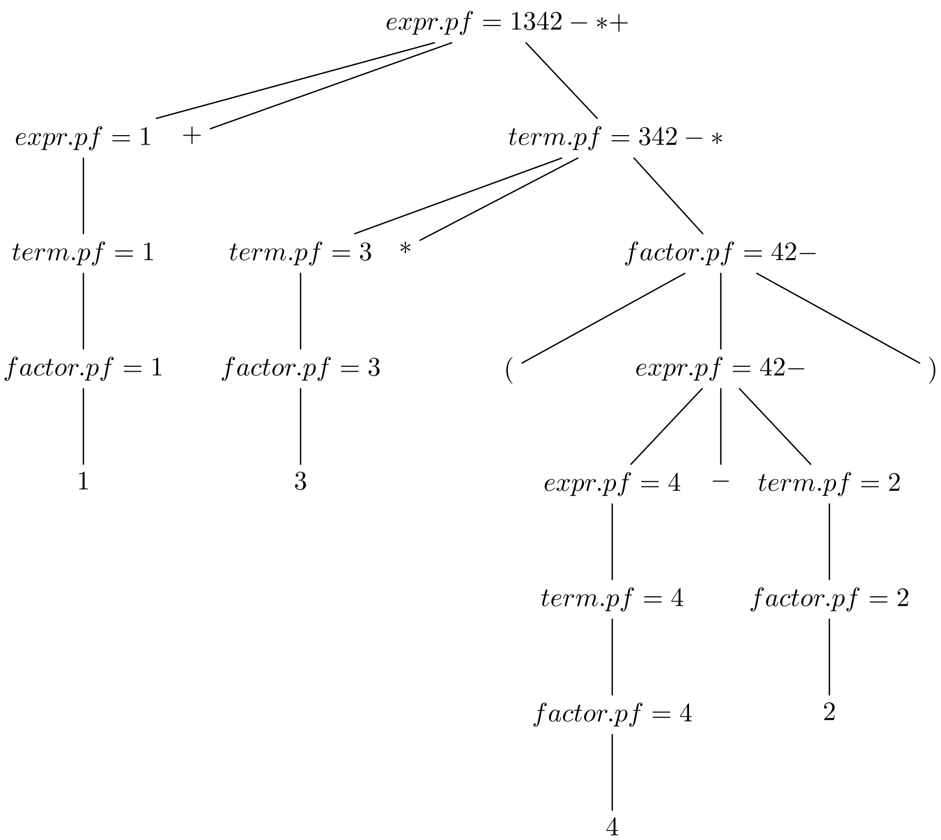
4. If E is of the form (E), the postfix notation of (E) is the postfix notation of E.

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.

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

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:

```
int lookahead;

void match(int token)
{
    if (token == lookahead)
    {
        // read next token into lookahead;
    }
    else
    {
        throw std::runtime_error("syntax error");
    }
}

void expr()
{
    // match an expression...
}

void stmt()
{
    match(IF); match('('); expr(); match(')'); stmt(); match(ELSE); stmt
        ();
}
```

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 $expr \to expr{+}term$, 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. Conside a nonterminal A with two productions

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

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

$$expr \to expr + term \,|\, term$$

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 $A \to A\alpha$ 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:

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

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:

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

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

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

[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:

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

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

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

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

$$P \rightarrow \alpha\, [abc]\, \beta \qquad\qquad \begin{aligned} P &\rightarrow \alpha R \beta \\ R &\rightarrow a \,|\, b \,|\, c \end{aligned}$$

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

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

become an iterative production:

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

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.

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

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 denoned 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` 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.

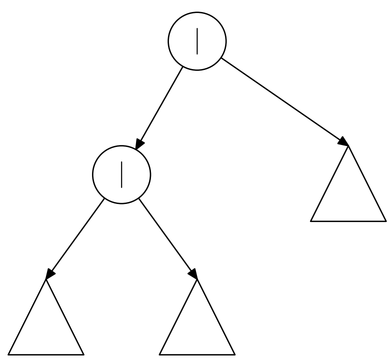
- The definition of a rule consists of nonempty sequence of *alternative* expressions:

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

  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



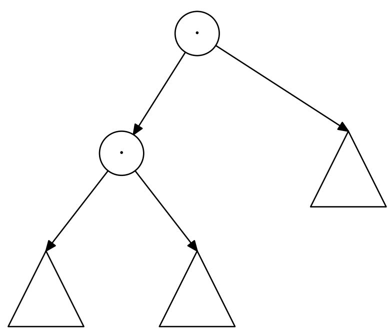
- Each alternative expression ω_i consists of catenation of expressions :

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

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



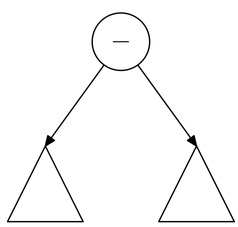
- A *difference* expression is denoted by α_i in a catenate expression $\alpha_1\alpha_2\cdots\alpha_k$. The difference expression consists of nonempty sequence of expressions separated by the $-$ symbol:

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

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



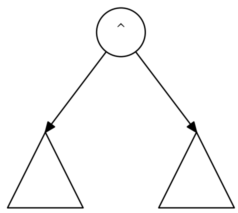
- An *xor* expression is denoted by β_i in a difference expression $\beta_1 - \beta_2 - \cdots - \beta_k$. The xor expression consists of nonempty sequence of expressions separated by the ˆ symbol:

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

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



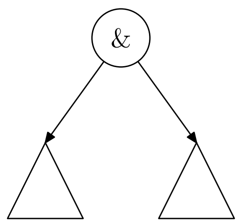
- An *intersection* expression is denoted by γ_i in an xor expression $\gamma_1\hat{}\gamma_2\hat{}\cdots\hat{}\gamma_k$. The intersection expression consists of nonempty sequence of expressions separated by the & symbol:

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

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:

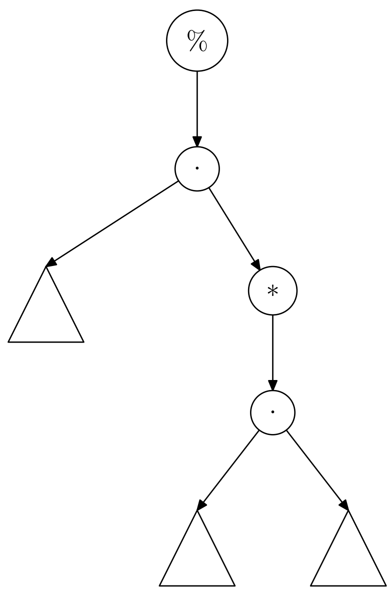
  $$\theta_1\,(\%\,\theta_2)?$$

  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 $\{\epsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}\}$ match expression $\mathbf{a}^*$.

     A *closure* node is a unary parsing node whose 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 $\mathbf{a}^+$.

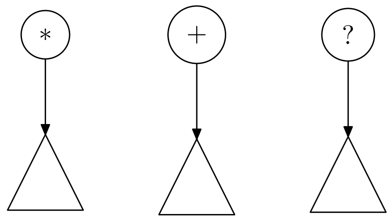
   A *positive* node is a unary parsing node whose subtree represents expression η.

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 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 expressions defined using the extended context-free notation are:

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

  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:

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

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.

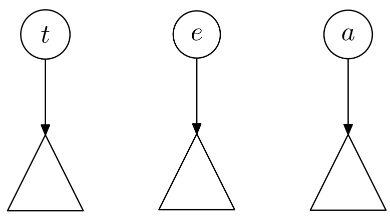
$$expectation \rightarrow \,'!'$$

- 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:

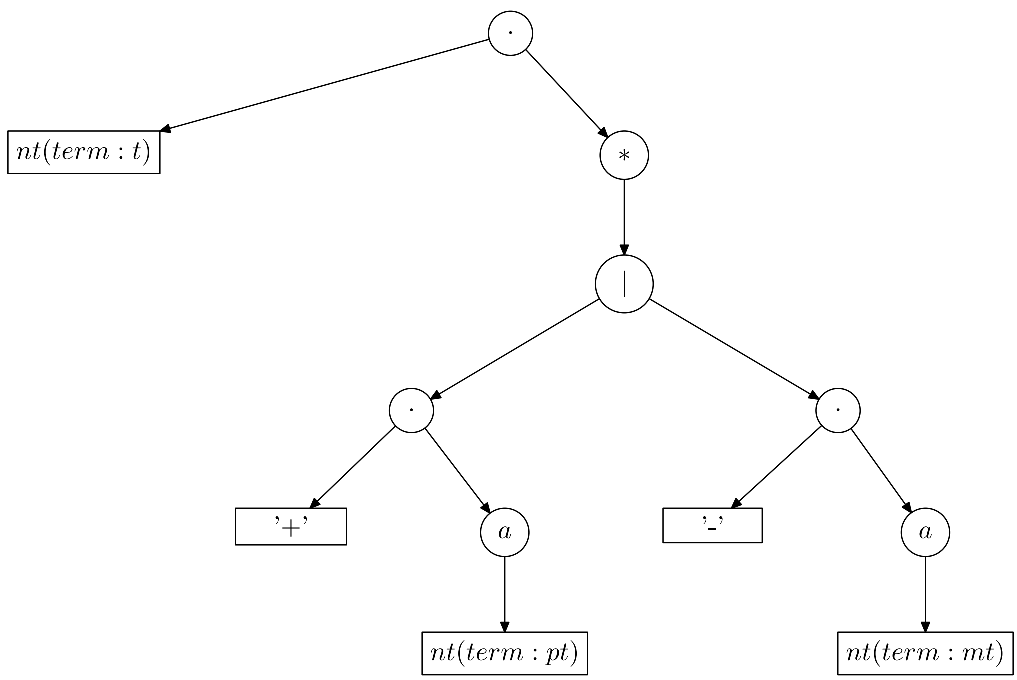
```
grammar PostfixTranslationGrammar
{
    expr: std::string
        ::= term:t{ value = t; }
        (   '+' term:pt{ value.append(pt).append(1, '+'); }
        |   '-' term:mt{ value.append(mt).append(1, '-'); }
        )*
        ;

    term: std::string
        ::= factor:f{ value = f; }
        (   '*' factor:tf{ value.append(tf).append(1, '*'); }
        |   '/' factor:df{ value.append(df).append(1, '/'); }
        )*
        ;

    factor: std::string
        ::= digit{ value = std::string(1, *matchBegin); }
        |   '(' expr{ value = expr; } ')'
        ;
}
```

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?\ \text{" ::= "}\ 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 $\mathbf{match}(\mathbf{true}, \mathit{length}(m_1) + \mathit{length}(m_2))$.

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

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 $\mathit{length}(m_2) < \mathit{length}(m_1)$, 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 $\mathit{length}(m_2) \geq \mathit{length}(m_1)$. Let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{false})$.

[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 $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{true}, 0)$, and let $first$ be **true**. Do following in a loop until loop exited:

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

   (b) If $first = \mathbf{true}$, 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 $\mathbf{match}(\mathbf{true}, length(m_1) + length(m_2))$, 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 $\mathbf{match}(\mathbf{true}, length(m_1) + length(m_2))$, 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 $\mathbf{match}(\mathbf{true}, 0)$.

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 $\mathbf{match}(\mathbf{true}, 1)$.

    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 $\mathbf{match}(\mathbf{false})$.

11. String node (Fig. 3.10). Let m be $\mathbf{match}(\mathbf{true}, 0)$. 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 $\mathbf{match}(\mathbf{true}, length(m) + 1)$.

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

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{true}, 1)$

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{false})$.

15. Empty node (Fig. 3.10). Let the result of parsing this node be $\mathbf{match}(\mathbf{true}, 0)$.

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 $\mathbf{match}(\mathbf{true}, 1)$.

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{true}, 1)$.

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{true}, 1)$.

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{true}, 1)$.

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{true}, 1)$.

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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 $\mathbf{match}(\mathbf{true}, 1)$.

    Otherwise let the result of parsing this node be $\mathbf{match}(\mathbf{false})$.

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.

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*