4. Lexical and Syntax Analysis

Why should we discuss the implementation of parts of a compiler?

- Syntax analyzers are based directly on the grammars discussed in Chapter 3.
- Lexical and syntax analyzers are needed in numerous situations outside compiler design including
  - program listing formatters
  - programs that compute the complexity of programs
  - programs that must analyze and react to the contents of a configuration file

4.1. Introduction

Lexical and Syntax Analysis are the first two phases of compilation as shown below.

![Figure 4.1 Lexical and Syntax Analysis](image)

Languages are designed for both phases

- For characters, we have the language of regular expressions to recognize tokens.
- For tokens, we have context free grammars to recognize syntactically correct programs.

Reasons for separating lexical analysis from syntax analysis from syntax analysis are:

1. Simplicity – Techniques for lexical analysis are less complex that those required for syntax analysis, so the lexical-analysis process can be simpler if it separate. Also, removing the low-level details of lexical analysis from the syntax analyzer makes the syntax analyzer both smaller and cleaner.
2. Efficiency – Although it pays to optimize the lexical analyzer, because lexical analysis requires a significant portion of total compilation time, it is not fruitful to optimize the syntax analyzer. Separation facilitates this selective optimization.
3. Portability – Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform dependent. However, the syntax analyzer can be platform independent. It is always a good practice to isolate machine-dependent parts of any software system.

4.2. Lexical Analysis

- A lexical analyzer is a pattern matcher.
- A lexical analyzer recognizes strings of characters as tokens.
- A token is a tuple (code, spelling)
  - code – an integer code is given to every unique pattern. Separate codes are assigned to all punctuation, every reserve word, all types of constants, and to identifiers.
  - spelling – the spelling is the actual string that was recognized. For example, the identifier, “result”. The string “result” is the spelling of the token.
Consider the following example of an assignment statement together with C++ declarations

\[ \text{result} = \text{oldsum} - \text{value} / 100; \]

Spelling | Symbol | Code
---|---|---
result | ID | 1
= | ASSIGN | 3
oldsum | ID | 1
- | MINUS | 4
value | ID | 1
/ | SLASH | 5
100 | INTLIT | 2
; | SEMICOLON | 6

Figure 4.2 Tokens of the statement \( \text{result} = \text{oldsum} - \text{value} / 100; \)

\begin{verbatim}
#define ID 1
#define INTLIT 2
#define ASSIGN 3
#define MINUS 4
#define SLASH 5
#define SEMICOLON 6
\end{verbatim}

Figure 4.3 C++ constant definitions that support tokens

- Lexical analyzers (scanners) extract lexemes (tokens) from a given input string.
- Lexical analyzers skip comments and blanks.
- There are three approaches to building a lexical analyzer:
  1. Write a formal description of the token patterns of the language using a descriptive language related to regular expressions. These descriptions are used as input to a software tool that automatically generates a lexical analyzer. The oldest and most accessible of these, name lex, is commonly included as part of UNIX systems.
  2. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
  3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.
- A state diagrams is a directed graph. The nodes of a state diagram are labeled with state names. The edges are labeled with the input characters that cause the transitions among the states.
- Finite state machines are collections of related state diagrams called finite automata.
- A class of languages called regular languages or regular expression can be translated to finite automata.
Figure 1. DFA accepting \((a|b)^*abb\)

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>1</td>
</tr>
<tr>
<td>(1)</td>
<td>1</td>
</tr>
<tr>
<td>(2)</td>
<td>1</td>
</tr>
<tr>
<td>(3)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Transition function for the DFA of Figure 1.

1. \(S = \{0,1,2,3\}\)
2. Table 1 shows the transition function move for the DFA of Figure 1.
3. \(\Sigma = \{a,b\}\)
4. \(s_0 = 0\)
5. \(F = \{3\}\)
4.3. The Parsing Problem

- Analyzing a sequence of tokens to determine if they form a sentence in the grammar of the programming language is called syntax analysis.
- Syntax analysis is often called parsing.

4.3.1. Introduction to Parsing

- Parsers for programming languages construct parse trees for given programs.
- There are two distinct goals of syntax analysis:
  1. The parser determines if the input program is syntactically correct.
     1.1. If an error is found, the parser generates a diagnostic message indicating the location of the error and a message that indicates why the program is not correct.
  2. The parser produces a parse tree of a syntactically correct program.
- There are two broad classes of parsers.
  1. **top-down**: A top-down parser attempts to construct the parse tree from the root down to its leaves.
  2. **bottom-up**: A bottom-up parser attempts to construct the parse tree from its leaves upward to the root.
- Parsing terminology.
  1. Terminal symbols – lowercase letters at the beginning of the alphabet. \((a, b, \cdots)\).
  2. Nonterminal symbols – uppercase letters at the beginning of the alphabet. \((A, B, \cdots)\).
  3. Terminals or nonterminals – uppercase letters at the end of the alphabet. \((W, X, Y, Z)\).
  4. Strings of terminals – lowercase letters at the end of the alphabet. \((w, x, y, z)\).
  5. Mixed strings (terminals or nonterminals) – lowercase Greek letters. \((\alpha, \beta, \gamma, \delta)\)
- Programming language terminology.
  1. Terminal symbols – terminal symbols are printed in bold. For example, **for**, **while**, ++, -.
  2. Nonterminal symbols – Nonterminal symbols are printed in italics. For example, *expression*, *term*, *factor*.

4.3.2. Top-Down Parsers

- A top-down parser traces or builds a parse tree in preorder. This corresponds to a leftmost derivation.
- The general form of a left sentential form that is \(xA\alpha\), recalling that \(x\) is a string of terminal symbols, \(A\) is a nonterminal, and \(\alpha\) is a mixed string,
- Because \(x\) contains only terminal symbols, \(A\) is the leftmost nonterminal in the sentential form, so it is the one that must be expanded to get the next sentential form in a leftmost derivation.
In the sentential form $xA\alpha$ a top-down parser must select one of the rules having $A$ on the left hand side. Given the following $A$-rules,

$$A \rightarrow bB$$
$$A \rightarrow cBb$$
$$A \rightarrow a$$

A top-down parser must use one of the foregoing rules to transform the left sentential form $xA\alpha$ to

$$xbBa$$
$$xcBba$$
$$xa\alpha$$

Recursive-descent parsers are the most common method of implementing a top-down parser.

A recursive-descent parser is coded directly from the BNF grammar and has one function or procedure for each nonterminal symbol.

Recursive-descent parsers employ LL algorithms. The first L is for a Left to right scan of the input. The second L is for a Leftmost derivation.

4.3.3. Bottom-Up Parsers

A bottom-up parser constructs a parse tree by beginning at the leaves and progressing toward the root.

Give a right sentential form $\alpha$, the parser must determine what substring of $\alpha$ is the RHS (right-hand side) of the rule in the grammar that must be reduced to its LHS (left-hand side) to produce the previous sentential form in the rightmost derivation.

Consider the following grammar and derivation

<table>
<thead>
<tr>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$aAc$</td>
</tr>
<tr>
<td>$A$</td>
<td>$aA$</td>
</tr>
<tr>
<td>$A$</td>
<td>$b$</td>
</tr>
</tbody>
</table>

1. Start with $aabc$, a string of terminal symbols
2. Replace terminal symbol $b$ with its LHS, $A$, yielding $aaAc$.
3. The LR parsing algorithm correctly selects the handle $aA$. The handle $aA$ is replaced by $A$ yielding $aAc$.
4. Finally $aAc$ is replaced by the goal symbol $S$ and parsing terminates.

4.3.4. The Complexity of Parsing

Parsing algorithms that work for any unambiguous grammar require $O(n^3)$ time.

By using a subset of context free grammars, the time complexity of parsing can be reduced to $O(n)$ time.
4.4. Recursive-Descent Parsing

4.4.1. The Recursive-Descent Parsing Process

- A recursive-descent parser is so named because it consists of a collection of
  subprograms, many of which are recursive, and it produces a parse tree in top-down
  order.
- EBNF is ideally suited for recursive-descent parsers.
- A recursive-descent parser has a subprogram for each nonterminal in the grammar.
- Consider the following EBNF for arithmetic expressions.

```
LHS    RHS
expression  →  term { (+ | -) term }
term      →  factor { (* | /) factor }
factor    →  id | intlit | ( expression )
```

```
void factor(bool get)
{
  ParsePrint("Enter factor");
  switch (Token()) {
    case ID:
    case INTLIT:
      Lex(); LexPrint(*o);
      break;
    case LPAREN:
      Lex(); LexPrint(*o);
      expr(false);
      Expected[0]=RPAREN;
      if (Token()!=RPAREN) throw ParseException(Expected,1,Token());
      Lex(); LexPrint(*o);
      break;
    default:
      ParsePrint("Exit factor");
      throw ParseException(Expected,3,Token());
      break;
  }
  ParsePrint("Exit factor");
}
```

Function factor
void term(bool get)
{
    ParsePrint("Enter term");
    factor(get);
    while (Token()==MUL_OP || Token()==DIV_OP) {
        Lex(); LexPrint(*o);
        factor(false);
    }
    ParsePrint("Exit term");
}

Function term

void expr(bool get)
{
    ParsePrint("Enter expr");
    term(get);
    while (Token()==ADD_OP || Token()==DIF_OP) {
        Lex(); LexPrint(*o);
        term(false);
    }
    ParsePrint("Exit expr");
}
• Example expression \((\text{sum}+47)/\text{total}\).

• Recursive-descent parse trace for the example expression

```
Token Code(8) Token Name(LPAREN) Token String="(
Enter expr
Enter term
Enter factor
Token Code(1) Token Name(    ID) Token String="sum"
Enter term
Enter factor
Token Code(4) Token Name(ADD_OP) Token String="+
Exit factor
Exit term
Token Code(2) Token Name(INTLIT) Token String="47"
Enter term
Enter factor
Token Code(9) Token Name(RPAREN) Token String=")"
Exit factor
Exit term
Exit expr
Token Code(7) Token Name(DIV_OP) Token String="/"
Exit factor
Token Code(1) Token Name(ID) Token String="total"
Enter factor
Token Code(0) Token Name(END) Token String=""
Exit factor
Exit term
Exit expr
```

Parse trace of the expression \((\text{sum}+47)/\text{total}\)
<table>
<thead>
<tr>
<th>Id</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>expression</td>
<td>→ term</td>
</tr>
<tr>
<td>2</td>
<td>expression</td>
<td>→ expression + term</td>
</tr>
<tr>
<td>3</td>
<td>expression</td>
<td>→ expression - term</td>
</tr>
<tr>
<td>4</td>
<td>term</td>
<td>→ factor</td>
</tr>
<tr>
<td>5</td>
<td>term</td>
<td>→ term * factor</td>
</tr>
<tr>
<td>6</td>
<td>term</td>
<td>→ term / factor</td>
</tr>
<tr>
<td>7</td>
<td>factor</td>
<td>→ ( expression )</td>
</tr>
<tr>
<td>8</td>
<td>factor</td>
<td>→ id</td>
</tr>
<tr>
<td>9</td>
<td>factor</td>
<td>→ intlit</td>
</tr>
</tbody>
</table>

Parse tree of \((\text{sum}+47)/\text{total}\)
### The LL Grammar Class

#### Parse tree of \((sum + 47)/total\)
• Eliminating direct left recursion.
  For each nonterminal, $A$,
  1. Group the $A$-rules as $A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ where none of the $\beta$’s begin with $A$.
  2. Replace the original $A$-rules with
     
     $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$
     
     $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon$ where $\varepsilon$ is the empty string.

• Apply the foregoing transformation rules to the canonical expression grammar given below.

\[
\begin{align*}
E & \rightarrow T \\
E & \rightarrow E + T \\
T & \rightarrow F \\
T & \rightarrow T * F \\
F & \rightarrow \text{id} \\
F & \rightarrow (E)
\end{align*}
\]

Applying the transformation rules:

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \\
E' & \rightarrow \varepsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' \\
T' & \rightarrow \varepsilon \\
E & \rightarrow E + T \\
F & \rightarrow (E)
\end{align*}
\]

• The parser must always be able to select the correct RHS based on the next token of input, using only the first token generated by the leftmost nonterminal in the current sentential form. This is called the pairwise disjointness test.

• The pairwise disjointness test is:
  For each nonterminal, $A$, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that
  
  $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$

  In other words, if a nonterminal $A$ has more than one RHS, the first terminal symbol that can be generated in a derivation for each of them must be unique to that RHS.
• Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If α ⇒ * ∈, then ∈ is also in FIRST(α).

To compute FIRST(X)

1. If X is a terminal, then FIRST(X) = \{X\}
2. If X is a nonterminal and X → Y₁ Y₂ ⋯ Yₖ is a production, then place α in FIRST(X) if for some i, α is in FIRST(Yᵢ), and ∈ is all of FIRST(Y₁), ⋯ , FIRST(Yᵢ₋₁); that is: Y₁ ⋯ Yᵢ₋₁ ⇒ * ∈. If ∈ is in FIRST(Yᵢ) for all j = 1,2, ⋯ , k, then add ∈ to FIRST(X). For example, everything in FIRST(Y₁) is surely in FIRST(X). If Y₁ does not derive ∈, then we add nothing more to FIRST(X) but if Y₁ ⇒ ∈, then we add FIRST(Y₂), and so on.
3. If X → ∈ is a production, then add ∈ to FIRST(X).

4.5. Bottom-Up Parsing
4.5.1. The Parsing Problem for Bottom-Up Parsers

Consider an abbreviated grammar for expressions and the sentence id+id*id

<table>
<thead>
<tr>
<th>Id</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>E + T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>T * F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>id</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>( E )</td>
</tr>
</tbody>
</table>

A bottom-up parse, a LR parse is given below. A LR parse is a Left-to-right scan of the input and a Rightmost derivation.

<table>
<thead>
<tr>
<th>Sentential form (rightmost derivation)</th>
<th>Id</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E + T</td>
<td>2</td>
<td>E</td>
<td>E + T</td>
</tr>
<tr>
<td>E + T * F</td>
<td>4</td>
<td>T</td>
<td>T * F</td>
</tr>
<tr>
<td>E + T * id</td>
<td>5</td>
<td>F</td>
<td>id</td>
</tr>
<tr>
<td>E + F * id</td>
<td>3</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>E + id * id</td>
<td>5</td>
<td>F</td>
<td>id</td>
</tr>
<tr>
<td>T + id * id</td>
<td>1</td>
<td>E</td>
<td>T</td>
</tr>
<tr>
<td>F + id * id</td>
<td>3</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>id + id * id</td>
<td>5</td>
<td>F</td>
<td>id</td>
</tr>
</tbody>
</table>
4.5.2. Shift-Reduce Algorithms

4.5.3. LR Parsers

1. The ACTION function takes as arguments a state \( i \) and a terminal \( a \) (\( $ \), the input endmarker). The value of \( \text{ACTION}[i,a] \) can have one of four forms:
   
   1.1. Shift \( j \) where \( j \) is a state. The action taken by the parser effectively shifts input \( a \) to the stack, but uses state \( j \) to represent \( a \).
   
   1.2. Reduce \( A \rightarrow \beta \). The action of the parser effectively reduces \( \beta \) on the top of the stack to head \( A \).
   
   1.3. Accept. The parser accepts the input and finishes parsing.
   
   1.4. Error. The parser discovers an error in its input and takes some corrective action.

2. We extend the GOTO function, defined on sets of items, to states: if \( \text{GOTO}[I_i,A]=I_j \), then \( \text{GOTO} \) also maps a state \( i \) and a nonterminal \( A \) to state \( j \).

![LR Parsing Program Diagram]

**Figure 1.** LR Parser Model

<table>
<thead>
<tr>
<th>left side</th>
<th>right side</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( E \rightarrow E + T )</td>
<td></td>
</tr>
<tr>
<td>2 ( E \rightarrow T )</td>
<td></td>
</tr>
<tr>
<td>3 ( T \rightarrow T*F )</td>
<td></td>
</tr>
<tr>
<td>4 ( T \rightarrow F )</td>
<td></td>
</tr>
<tr>
<td>5 ( F \rightarrow (E) )</td>
<td></td>
</tr>
<tr>
<td>6 ( F \rightarrow id )</td>
<td></td>
</tr>
</tbody>
</table>

*Table 1. Set of productions expressions*

let \( a \) be the first symbol of \( wS \)

while (1) {
    let \( s \) be the state on top of the stack;
    if (\( \text{ACTION}[s,a]==\text{shift} \ t \)) {
        push \( t \) onto the stack
        let \( a \) be the next input symbol;
    } else if (\( \text{ACTION}[s,a]==\text{reduce} \ A \rightarrow \beta \)) {
        pop \( \beta \) symbols off the stack;
        let state \( t \) now be on top of the stack;
        push \( \text{GOTO}[t,A] \) onto the stack
        output the production \( A \rightarrow \beta \);
    } else if (\( \text{ACTION}[s,a]==\text{accept} \)) break; //parsing is done
    else error();
}
### STACK SYMBOLS INPUT ALGORITHM ACTION

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td>id</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(2)</td>
<td>0 5</td>
<td>id</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(3)</td>
<td>0 3</td>
<td>F</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(4)</td>
<td>0 2</td>
<td>T</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(5)</td>
<td>0 2 7</td>
<td>T</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(6)</td>
<td>0 2 7 5</td>
<td>T</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(7)</td>
<td>0 2 7 10</td>
<td>T</td>
<td>*</td>
<td>F</td>
<td>$</td>
</tr>
<tr>
<td>(8)</td>
<td>0 2</td>
<td>T</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(9)</td>
<td>0 1</td>
<td>E</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(10)</td>
<td>0 1 6</td>
<td>E</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(11)</td>
<td>0 1 6 5</td>
<td>E</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
<tr>
<td>(12)</td>
<td>0 1 6 3</td>
<td>E</td>
<td>*</td>
<td>F</td>
<td>$</td>
</tr>
<tr>
<td>(13)</td>
<td>0 1 6 9</td>
<td>E</td>
<td>*</td>
<td>T</td>
<td>$</td>
</tr>
<tr>
<td>(14)</td>
<td>0 1</td>
<td>E</td>
<td>*</td>
<td>id</td>
<td>$</td>
</tr>
</tbody>
</table>

### STATE ACTION GOTO

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>+</td>
<td>$</td>
<td>1</td>
<td>E</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td>*</td>
<td>(</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s7</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>s4</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r6</td>
<td>r6</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r7</td>
<td>s4</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r5</td>
<td>r5</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r6</td>
<td>r6</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
### Lexical and Syntax Analysis

#### Chapter 4

<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s7</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>5</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>8</td>
<td>s6</td>
<td>s11</td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>s7</td>
</tr>
<tr>
<td>10</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td>r5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STACK</th>
<th>SYMBOLS</th>
<th>INPUT</th>
<th>ALGORITHM</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td>id + id * id $</td>
<td>ACTION[0,id]=s5 shift</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>0 5</td>
<td>id</td>
<td>+ id * id $</td>
<td>ACTION[5,+]=r6 GOTO[0,F]=3 reduce by F → id</td>
</tr>
<tr>
<td>(3)</td>
<td>0 3</td>
<td>F</td>
<td>+ id * id $</td>
<td>ACTION[3,+] r4 GOTO[0,T]=2 reduce by T → F</td>
</tr>
<tr>
<td>(4)</td>
<td>0 2</td>
<td>T</td>
<td>+ id * id $</td>
<td>ACTION[2,+] r2 GOTO[0,E]=1 reduce by E → T</td>
</tr>
<tr>
<td>(5)</td>
<td>0 1</td>
<td>E</td>
<td>+ id * id $</td>
<td>ACTION[1,+] s5 shift</td>
</tr>
<tr>
<td>(6)</td>
<td>0 1 6</td>
<td>E +</td>
<td>id * id $</td>
<td>ACTION[6,id]=s5 shift</td>
</tr>
<tr>
<td>(7)</td>
<td>0 1 6 5</td>
<td>E + id</td>
<td>* id $</td>
<td>ACTION[5,*]=r6 GOTO[6,F]=3 reduce by F → id</td>
</tr>
<tr>
<td>(8)</td>
<td>0 1 6 3</td>
<td>E + F</td>
<td>* id $</td>
<td>ACTION[3,*]=r4 GOTO[6,T]=9 reduce by T → F</td>
</tr>
<tr>
<td>(9)</td>
<td>0 1 6 9</td>
<td>E + T</td>
<td>* id $</td>
<td>ACTION[9,*]=s7 shift</td>
</tr>
<tr>
<td>(10)</td>
<td>0 1 6 9 7</td>
<td>E + T *</td>
<td>id $</td>
<td>ACTION[7,id]=s5 shift</td>
</tr>
<tr>
<td>(11)</td>
<td>0 1 6 9 7 5</td>
<td>E + T * id</td>
<td>$</td>
<td>ACTION[5,$]=r6 GOTO[7,F]=10 reduce by F → id</td>
</tr>
<tr>
<td>(12)</td>
<td>0 1 6 9 7 10</td>
<td>E + T * F</td>
<td>$</td>
<td>ACTION[10,$]=r3 GOTO[7,T]=9 reduce by T → T * F</td>
</tr>
<tr>
<td>(13)</td>
<td>0 1 6 9</td>
<td>E + T</td>
<td>$</td>
<td>ACTION[9,$]=r1 GOTO[0,E]=1 reduce by E → E + T</td>
</tr>
<tr>
<td>(14)</td>
<td>0 1</td>
<td>E</td>
<td>$</td>
<td>ACTION[1,$]=acc accept</td>
</tr>
</tbody>
</table>

\((s_0s_1\cdots s_m, a_i, a_{i+1} \cdots a_n)\)

\(X_1X_2\cdots X_m a_i a_{i+1} \cdots a_n\)