CS3410: Software Engineering Lab
LR(1) parser for JavaCC

Aravind S CS11B033
Akshayaram S CS11B057
Srinivasan R CS11B059
Ramnandan SK CS11B061
Adit Krishnan CS11B063

April 28, 2014

Contents

Chapter 1

Introduction

1.1 Context Free Grammars

A Context Free Grammars (CFG) consists of Terminals, Non-terminals, Start Symbol and Productions.

1. Terminals are the basic symbols from which strings of the language are formed. Terminals are also called as tokens and correspond to the tokens that are returned by the lexical analyzer.

2. Non-Terminals are syntactic variables that denote a set of strings. Non-terminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. A non-terminal is distinguished as the start symbol and the set of strings that are derived from the start symbol comprises of the language.

4. The productions of a grammar specify the manner in which terminals and non-terminals are combined to form strings. In a CFG:

   - A single non-terminal to the left side of the production which is called as the head or the lhs.
   - The body or the rhs which comprises of zero or more non-terminals and terminals.

1.1.1 Derivations

A derivation is composed of the set of productions that must be applied to get a given string belonging to the language starting at the start symbol.

A *leftmost derivation* is one in which the leftmost non-terminal is replaced with the right hand side of the production involving the non-terminal.

A *rightmost derivation* is one in which the rightmost non-terminal is replaced with the right hand side of the production involving the non-terminal.

1.2 Lexical Analyzers and Parsers

Parsers and lexical analyzers are the first two components that are present in the front end of any compiler. *Lexical analyzer* reads a stream of characters from the source program and groups the characters into meaningful sequences called as *tokens* which is passed to the next phase of the compiler which is the parser.

Parser or the syntax analyzer uses the tokens produced by the lexical analyzer to produce a tree-like intermediate representation which depicts the grammatical structure of the token stream. A typical representation is that of a syntax tree in which each interior node represents an operation and the leaves represent the operands or the arguments.

1.2.1 Top Down Parsing

Top down parsing can be viewed as the problem of generating the parse tree for the input string in the depth first order starting at the root. The class of grammars for which one could construct predictive top down parsers with k look ahead symbols is called as $LL(k)$ grammars where first L refers to the left to right scanning, the second L stands for leftmost derivation and k refers to the number of lookahead tokens used in decision making.

$LL(k)$ Grammars

A Context Free Grammar is called $LL(k)$ if and only if for all non-terminals A and each distinct pair of production $A \rightarrow \beta$ and $A \rightarrow \gamma$ satisfy the condition $First_k(\beta) \cap First_k(\gamma) = \phi$ where $First_k(\alpha)$ the set of k terminals that occur in the beginning in all strings derived from non-terminal α.

It is a well known fact that predictive parsers i.e recursive decent parsers can be constructed for $LL(k)$ grammars without backtracking and requiring k lookahead tokens. $LL(k)$ parsers are easy to generate and are very efficient parsers can be generated for $LL(k)$ grammars. But it suffers from the following drawbacks:

1. It cannot parse any left recursive grammar.
2. It cannot parse any ambiguous grammar (An ambiguous grammar is one in which there exists a string in the language generated by the grammar which has at least two different parse trees)
3. Some languages have no $LL(k)$ grammars.

1.2.2 Bottom Up Parsing

A bottom up parsing corresponds to constructing a parse tree for an input from the leaves and working towards the root. One can think of bottom up parsing as a process of reducing a string to the start symbol of the grammar. The most common grammars that can be parsed with a bottom up parser is called as the $LR(k)$ parsers where first L refers to left to right scanning of input, R refers to constructing reverse rightmost derivation and k is the number of lookahead symbols.

$LR(k)$ Grammars

A grammar is said to be $LR(k)$ if given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \gamma_k = w$$

we can determine for each right sentential form in the derivation

- The handle to used
- The production to be used for reduction

by scanning γ_i from left to right and scanning at most k symbols beyond right end of the handle γ_i.

We will look into more details of $LR(k)$ grammar in the next chapter.

1.3 JavaCC

JavaCC is a parser generator and a lexical analyzer written in Java programming language. It is a open source project and is similar to yacc in that it generates a parser from the formal grammar written in Extended Backus-Naur form (EBNF). JavaCC generates top down parsers and hence limited to parsing only $LL(k)$ grammars and hence cannot parse left recursive grammars. Some of the highlights of JavaCC are:

1. JavaCC enjoys a large user community and is by far the most widely used parser generator for Java applications.
2. An important feature of JavaCC is that both the lexical specification such as token, regular expressions, strings and the BNF for the grammar can be specified in the same file unlike the flex-Bison combination where the lexical specifications and the grammar specification are written in separate files.
3. JavaCC comes with JJtree which is an extremely powerful tree processor. It converts the given grammar file to a parse tree with internal nodes indicating the non-terminals and the leaves indicating the terminals.
4. By default JavaCC generates an $LL(1)$ parser but it is highly customizable and one can specify the number of lookahead tokens (k) to be used just before running it on a specific grammar file.

JavaCC uses $LL(k)$ parser and hence cannot parse a lot of languages that can be parsed only by a $LR(1)$ parser (for example left recursive grammars). Hence, we wanted to write a $LR(1)$ parser for a restrictive choice of grammars for JavaCC.

Chapter 2

LR(1) parsers

This chapter gives a brief introduction about $LR(1)$ parsing algorithms and constructing the $LR(1)$ parser table that we used in writing a $LR(1)$ parser for JavaCC. Before we go into the construction, we wish to specify a few definitions and notations that we will be using throughout the chapter. We have referred extensively to [1] for the algorithms and the techniques explained in this chapter.

2.1 Preliminaries

Bottom up parsing or $LR(1)$ parsing is the process of *reducing* a string w to start symbol by applying a series of production rules. At each *reduction* step, a specific substring matching the rhs of a production is replaced with the a non-terminal corresponding to the lhs or the head of the production.

Bottom up parsing during a left to right scan of the input constructs a rightmost derivation in reverse. A *handle* is a substring that matches the body of a production and whose reduction represents one step in the reverse rightmost derivation.

$First(\alpha)$ is defined as the set of terminals that begin strings derived from α. If $\alpha \Rightarrow \epsilon$, then $\epsilon \in First(\alpha)$.

2.2 Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom up parsing in which a stack holds a set of grammar symbols and buffer contains the rest of the input that is to be parsed.

Informally, during a left to right scan of the input the parser shifts 0 or more symbols to the top of the stack until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of appropriate production. The parser repeats this until it has detected an error or the stack consists of the start symbol and the buffer is empty.

A shift-reduce parser has four canonical actions:

- *Shift*: Shift the next input on top of the stack.
- *Reduce*: Right end of the handle is on top of the stack. Locate the left end of the handle within the stack. Pop the handle off the stack and push the appropriate non-terminal into the stack.
- *Accept:* Accepts the successful parsing of the given string.
- *Reject:* Discover a syntax error and call the error recovery routine.

Algorithm 1 Shift-Reduce Parser

```
Require: Goto table, action table and string w.
  push $s_0$
  $token \leftarrow next\_token()$
  while true do
    $s \leftarrow$ top of stack
    if $action[s, token]$ = "shift $s_i$" then
      push $s_i$
      $token \leftarrow next\_token()$
    else if $action[s, token]$ = "reduce $A \rightarrow \beta$" then
      pop $|\beta|$ states from the stack
      $s'$ = top of the stack
      push $goto[s', A]$
    else if $action[s, token]$ = "accept" then
      return
    else
      error()
    end if
  end while
```

2.3 Computing First of a non-terminal

We now give an algorithm for computing the first set of a non-terminal.

2.4 $LR(1)$ items

A $LR(1)$ item is pair $[\alpha, \beta]$ where

- α is a production in G with a $\bullet$ at some position in the rhs indicating how much of the input has been read.
- β is the lookahead symbol.

Algorithm 2 Computing $First(X)$

```
if $X$ is terminal then
  $FIRST(X) = X$
end if
if $X \Rightarrow \epsilon$ then
  Add $\epsilon$ to $First(X)$.
end if
if $X \Rightarrow Y_1Y_2...Y_k$ then
  Add $First(Y_1) - \epsilon$ to $FIRST(X)$
  for i = 1 to k do
    if $Y_1Y_2...Y_i \Rightarrow \epsilon$ then
      Add $First(Y_{i+1}) - \epsilon$ to $FIRST(X)$
    end if
  end for
  if $Y_1Y_2...Y_k \Rightarrow \epsilon$ then
    Add $\epsilon$ to $FIRST(X)$
  end if
end if
```

2.4.1 $Closure1(I)$

Given an item $[A \rightarrow \alpha \bullet B\beta, a]$ its closure contains the item and any other items that can generate legal substrings to follow α. We now provide the algorithm for computing the closure.

Algorithm 3 Computing $Closure1(I)$

```
repeat
  if $[A \rightarrow \alpha \bullet B\beta, a] \in I$ then
    Add $[B \rightarrow \bullet\gamma, b]$ to $I$ where $b \in First(\beta a)$
  end if
until no more items can be added to $I$
```

2.4.2 $Goto1(I)$

Let I be a set of $LR(1)$ items and X be a grammar symbol. $Goto1(I, X)$ is the closure of $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X\beta] \in I$

We now give the algorithm for computing the $Goto1(I, X)$

Algorithm 4 $Goto1(I, X)$

```
Let $J$ be the set of items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X\beta] \in I$.
return $Closure1(J)$
```

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1985.