Indian Institute of Technology, Kanpur

MTH 401 Project

LR(1) Parsers
Theory and Implementation

Siddharth Vishwanath
10712

Shivendra Kumar

November 2012

.

Preface

This project gives an outline of the use of Parsers in compilers. It starts off with the basic definition and uses of parsers with repsecct to CFGs and then goes on to demonstrate the mechanism of the LR Parsers.

It aims at providing the theory of parsers side-by-side with the algorithms used to implement it.

The project also has a program that implements the concept of the LR Parser, written in Python.

This project is done as a part of the course MTH 401 : Theory of computation, Indian Insitute of Technology Kanpur.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is a Parser?

A natural language parser is a program that works out the grammatical structure of sentences, for instance, which groups of words go together (as "phrases") and which words are the subject or object of a verb.

Parsing, or, more formally, *syntactic analysis*, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar.

In computer technology, a parser is a program, usually part of a compiler, that receives input in the form of sequential source program instructions, interactive on-line commands, markup tags, or some other defined interface and breaks them up into parts (for example, the nouns (objects), verbs (methods), and their attributes or options) that can then be managed by other programming (for example, other components in a compiler). A parser is usually used to check to see that all input has been provided that is necessary.

Figure 1.1 represents the analysis of the flow of a string in a Parser.

1.2 Context Free Grammars

Context-free grammars are a recursive representation of Context-free languages which are a larger class of Regular Languages. CFGs play a substantial role in compiler technology since the 1960s. They turned the implementation of parsers from a time-consuming ad-hoc implementation task to a routine job that can be done in an afternoon.

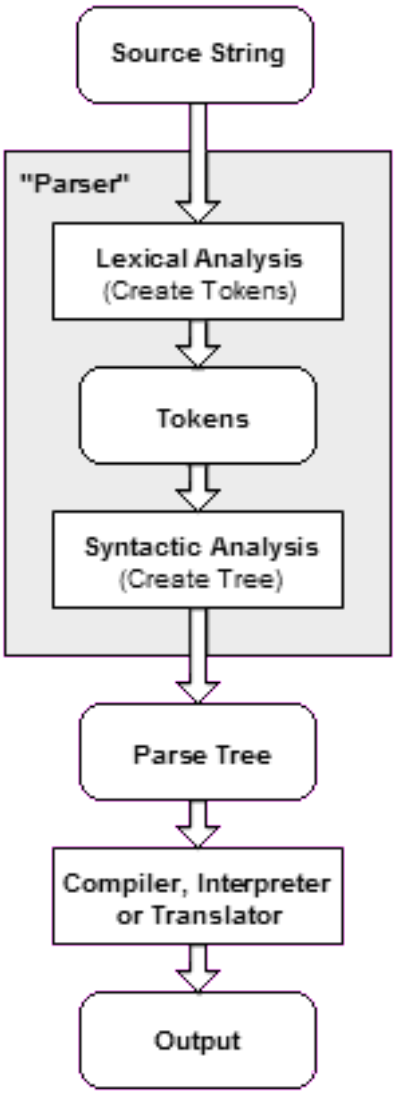

FIGURE 1.1: Flow of data in the typical parser

Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out.

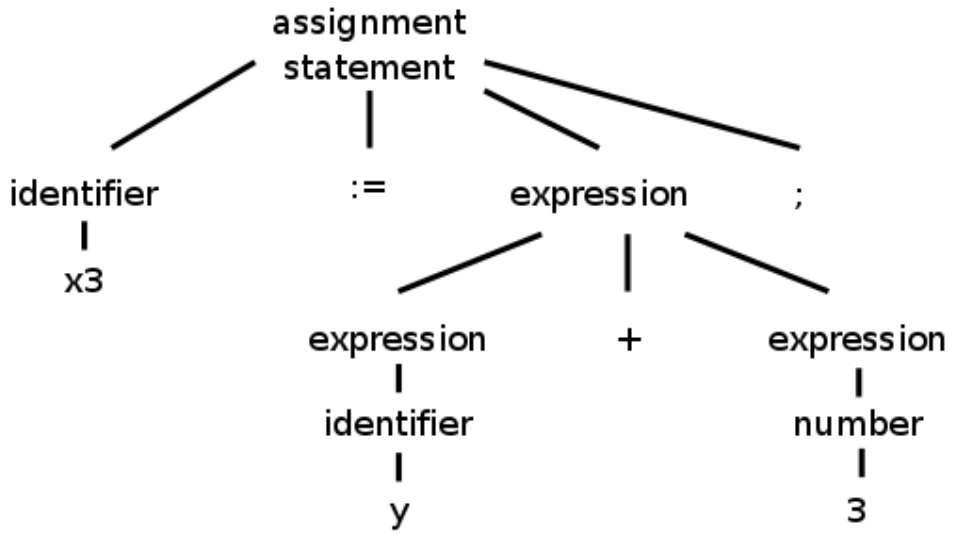

FIGURE 1.2: Example of a parse tree

1.3 Parse Trees

Parse trees are a tree-representation of the derivations that we encounter in a CFG. Parse trees clearly show how the symbols of a terminal string are grouped into substrings, each of which belong to the language.

When used in a compiler, the parse tree represents the source program. In computers, the tree structure of the source program facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

Figure 1.2 shows a parse-tree for an assignment operation in a typical computer program.

1.4 Bottom-Up Parsing

Bottom-up parsing is a parsing technique in which the abstract-syntax tree is in the bottom up fashion. The bottom-up name comes from the concept of a parse tree. A bottom-up parse discovers and processes the tree starting from the bottom left end and incrementally works its way upwards and rightwards. The tree may be merely implicit in the parser's actions. the opposite of this parsing technique is the top-down parsing, in which the input's overall structure is decided first before moving down the abstract-syntax tree, leaving the lowest level small details to the last.

1.4.1 Advantages of Bottom-up Parsing

1. Precedence, Associativity, Left-recursion are not required in Bottom-up parsing.
2. It involves less transformations and rewriting of existing grammar.
3. Bottom-up parsers accept a wider class of languages.

There are two kinds of *bottom-up* parsers.

1. LR parsers
   (a) SLR Parsers
   (b) LR(k) Parsers
   (c) LALR(k) Parsers
2. Operator-Precedence Parsers

Our following discussion will focus on the first type of Parsers. The **LR parsers**.

1.5 LR Parsers

In computer science, LR parsers are a type of bottom-up parsers that efficiently handle deterministic context-free languages in guaranteed linear time. The LALR parsers and the SLR parsers are common variants of LR parsers. LR parsers are often mechanically generated from a formal grammar for the language by a parser generator tool. They are very widely used for the processing of computer languages, more than other kinds of generated parsers.

LR parsers are deterministic; they produce a single correct parse without guesswork or backtracking, in linear time. This is ideal for computer languages. But LR parsers are not suited for human languages which need more flexible but slower methods. Other parser methods that backtrack or yield multiple parses may take O(n^2) or O(n^3) in the worst case scenario.

Chapter 2

LR Parsers

2.1 LR(k) Parsing

The technique used by LR parsers to parse a large class of CFGs is called LR(k) parsing. Here,

- The "**L**" stands for Left-to-Right scanning of the input
- The "**R**" stands for constructing a rightmost derivation in reverse and,
- The "**k**" stands for number of input symbols of look-ahead that are used for making the parsing decisions.

When "k" is omitted, it is assumed to be 1.

2.1.1 Advantages of LR Parsing

1. LR parsers can be constructed to recognize virtually all programming language construct for which context free grammars can be written.
2. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. LR grammars can describe more languages than LL grammars.

2.2 Parsers

As discussed earlier, there are three types of LR Parsers that employ the bottom-up method of parsing a string in a given CFG.They are

1. LR parsers
   (a) SLR Parsers
   (b) LR(k) Parsers
   (c) LALR(k) Parsers

The main drawback is that it is too much work to construct an LR parser by hand for a typical programming language grammar. For this, we need a specialized tool — and LR parser generator. Fortunately, many such generators are available such as YACC. [1] With such generators, one can write a CFG and have the generator automatically produce a parser for that grammar. We replicate this concept in our program, with the restriction that the CFG must be in the Chomsky Normal Form (CNF).

In our discussion, we start of with the parsing algorithm in SLR parsers and use the concepts developed in the same as a basis for our discussion of LR(1) parsers. So, we first begin with the SLR algorithm for constructing a parsing table for a CFG. This is also called the LR(0) parsing technique. We discuss its drawbacks with certain grammars and then present the parsing technique with a look-ahead symbol. This is called the LR(1) or canonical LR technique.

2.3 LR Parsing Algorithm

Figure 2.1 schematically represents an LR Parser. It typically consists of :

1. An input string
2. An output
3. A stack
4. A driver program and,
5. A parsing table which is divided into the ACTION and GOTO parts.

[1] (*Introduction to Automata Theory, Languages and Computation*, Hopcroft, Motwani, Ullman)

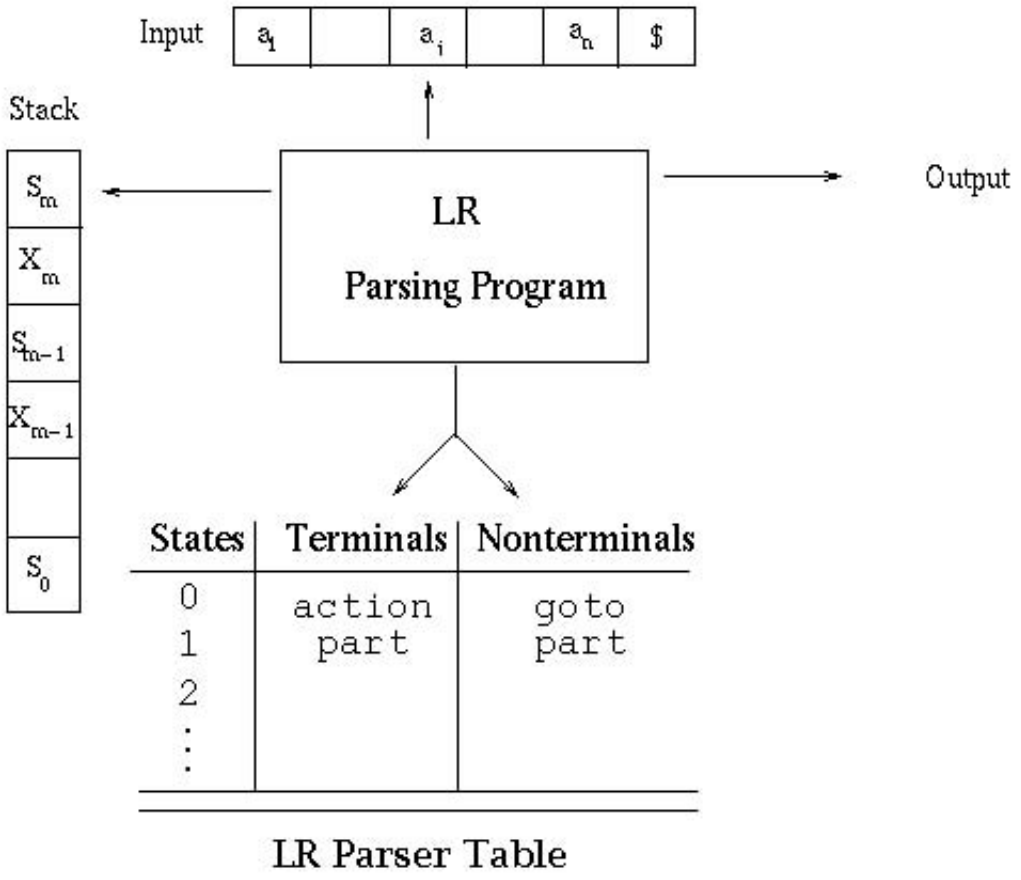

FIGURE 2.1: A typical LR parser

The method used involves the following:

2.3.1 Bottom-up Parsing

The method adopted is bottom up parsing. A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). a general style of bottom-up parsing known as shift reduce parsing has been adopted for LR grammars.

2.3.2 Reduction

This is the process of "reducing" a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production. A reduction is the reverse of a step in a derivation.

2.3.3 Handles

Bottom-up parsing during a left-to-right scan of the input constructs a right most derivation in reverse. Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation ,that is , when there is string w of a grammar.

2.3.4 Shift-Reduce Action

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. The handle always appears at the top of the stack just before it is identified as the handle. We use $ to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string w is on the input with stack as $ and string w as w$. During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto 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 the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

So to summarize our discussion so far,

- Top-down parsers build a parse tree from root to leaves
- Bottom-up parsers build a parse tree from leaves to root
- LR(1) parsers :
  - scan the input from left to right
  - build a rightmost derivation in reverse
  - use a single token lookahead to disambiguate
  - have a simple, table-driven, shift-reduce skeleton
  - use shift-reduce action and encode grammatical knowledge in ACTION-GOTO tables

Now, we go ahead to discuss some functions that help us develop the parsing algorithm.

2.3.5 LR(1) Items

An item set is the list of production rules, which the currently processed symbol might be part of. An item set has a one-to-one correspondence to a parser state, while the items within the set, together with the next symbol, are used to decide which state transitions and parser action are to be applied. The general form a LR(1) item is [A→ $\alpha \bullet \beta$,a],where A→ $\alpha\beta$ is a production and a is terminal or right $ endmarker. The dot represents how of an item we have seen in a given state. The second component is called the lookahead of the item. The lookahead has no effect in an item of the form [A→ $\alpha \bullet \beta$,a] where β is not ε, but item of the form [A→ $\alpha \bullet$,a] calls for reduction by A→ α only if the next symbol is a.

The • indicates how much of an item we have seen at a given state in the parse.

> [A → $\bullet XYZ$] indicates indicates that the parser is looking for a string that can be derived from XYZ
>
> [A → $XY \bullet Z$] indicates that the parser has seen a string derived from XY and is looking for one derivable from Z

The canonical set of LR(1) items have set of items:

- derivable from [S' → • S ,$] and
- that can derive final configuration.

2.3.6 FIRST

For a string of grammar symbols α, define FIRST(α) as

- The set of terminal symbols that begin strings derived from α
- If $L \Rightarrow \varepsilon$, then $\varepsilon \in \text{FIRST}(\alpha)$

FIRST(α) contains the set of tokens that are valid in the initial position in α.

Algorithm to build FIRST(X):

1. IF X is a terminal, FIRST(X) is {X}
2. IF X $\rightarrow \varepsilon$, then $\varepsilon \in$ FIRST(X)
3. IF X $\rightarrow Y_1 Y_2 \ldots Y_k$ then put FIRST(Y_1) in FIRST(X)
3. IF X is non-terminal and X $\rightarrow Y_1 Y_2 \ldots Y_k$, then
4. DO $a \in$ FIRST(X) if $a \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \leq j \leq i$

2.3.7 CLOSURE(I)

The closure operation is construction of set of items through two rules:

1. initially add every item in I to closure (I)
2. if A$\rightarrow \alpha \bullet B\beta$ is in closure (I) and B $\rightarrow \gamma$ is a production ,then add B$\rightarrow \bullet\gamma$ to I if not there
3. We apply these rule until no more items can be added.

2.3.7.1 Algorithm to compute CLOSURE(I)

To compute closure(I)

```
function closure(I)
   add = 1
   while (add ≠ 0)
      add = 0
      for each item [A ::= α • Bβ, a] ∈ I,
         each production B ::= γ ∈ G',
         and each terminal b ∈ FIRST(βa),
         if [B ::= •γ, b] ∉ I then
            add [B ::= •γ, b] to I
            add = 1
      return I
```

FIGURE 2.2: Algorithm to implement closure operation

2.3.8 GOTO

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

If I is the set of valid items for some viable prefix γ, then GOTO(I,X) is the set of valid items for the viable prefix γX. GOTO(I,X) represents state after recognizing X in state I.

2.3.8.1 Algorithm to compute GOTO(I,X)

To compute goto(I, X)

```
function goto(I, X)
  J ← set of items [A ::= αX • β, a]
    such that [A ::= α • Xβ, a] ∈ I
  J' ← closure(J)
  return J'
```

FIGURE 2.3: Algorithm to implement GOTO operation

2.3.9 Augmented Grammar

We start the construction of the collection of sets of LR(1) items with the item [S'$\rightarrow \bullet S$, \$], where

1. S' is the start symbol of the augmented grammar G'
2. S is the start symbol of G, and \$ is the right end of string marker

To compute the collection of sets of the LR(1) items, we first augment the given CFG and them implement the algorithm described in figure 2.4

2.3.9.1 Working with Augmented Grammar

To compute the collection of sets of LR(1) items

```
procedure items(G')
  C ← {closure({[S' ::= •S, eof]})}
  repeat
    new_item ← false
    for each set of items I in C and
      each grammar symbol X such that
      goto(I, X) ≠ ∅ and
      goto(I, X) ∉ C
        add goto(I, X) to C
        new_item ← true
    endfor
  until (new_item = false)
```

FIGURE 2.4: Algorithm to compute the collection of sets of LR(1) items

2.3.10 FOLLOW

For a non terminal A we define FOLLOW(A) as:

- the set of terminals that appear immediately to right of A in some sentential form.
- a terminal symbol has no FOLLOW set.

The method of building the FOLLOW set is as follows:

1. Place $ in sc follow(S), where S is the start symbol and $ is the input right- end-marker.
2. If there is a production A $\rightarrow \alpha B \beta$,then everything in FIRST(β) except for ε is placed in FOLLOW(B).

2.4 LR(1) ACTION-GOTO Tables

The ACTION-GOTO table for the LR(1) parser serves as a reference to the parser at every move. The ACTION-GOTO table basically represents a DFA that instructs the parser to perform a shift or reduce action when it undergoes a transition from one state to another.

The notations used here are as follows:

- **$** represents the end of an input string
- **sn** tells the parser to shift to state I_n
- **rk** tells the parser to reduce by rule k described in the CFG
- **acc** indicates that the input string is accepted
- **j** in the GOTO part of the table represents GOTO(j)

With this, we are now ready to describe the method for constructing the LR(1) ACTION-GOTO Table.

2.4.1 The algorithm

1. Construct the collection of sets of LR(1) items for G'
2. State i of the parser is constructed from I_i
   (a) if [A $\rightarrow \alpha\bullet a\beta$, b]$\in I_i$, and GOTO($I_i$,A)=$I_j$ then set set ACTION[i,a] to ''shift j'' or sj. (here, a must be a terminal)
   (b) if [A $\rightarrow \alpha\bullet$, a]$\in I_i$, then set set ACTION[i,a] to ''reduce A$\rightarrow \alpha$'' or rk. (here, k represents the k^{th} rule of the CFG)
   (c) if [S' $\rightarrow S\bullet$, $]$\in I_i$, then set action[i,$] to ''acc''
3. if GOTO(I_i,A)=I_j, the set GOTO[i,a] in the table to j.
4. All other entries in the ACTION-GOTO Table are set to ''error''
5. I_0, the initial state of the parser, is the closure of the item [S'$\rightarrow \bullet S$,$]. All other states are formed by the above procedure of constructing a DFA to represent the parsing action on each item in I_0 recursively.

2.5 Example of LR(1) Parsing

2.5.1 The CFG

Consider the CFG represented by :

- Terminals { $, ; , id, := , +}
- Non-terminals {S', S, A, E}
- Start Symbol = S'
- Productions :
  0. S' → S $
  1. S → S ; A
  2. S → A
  3. A → E
  4. A → id := E
  5. E → E + id
  6. E → id

2.5.2 Computing the closure

I_0 = CLOSURE({ [S'→ •S, $] })

For, [S'→ •S, $], we have B = S and $\beta = \varepsilon$
So, S → A S → S ; A FIRST($) = $

and, we have the items [S→ •A, $] [S→ •$S$; A, $]

For [S→ •A, $], we have B = A and $\beta = \varepsilon$
So, A → E A → id := E FIRST($) = $

and, we have the items [A→ •E, $], [S→ •$id := E$, $].

And, similarly we compute I_0 = CLOSURE({[S'→ •S, \$]}) to be :

```
[S' → . S, $],
[S → . A, $],
[S → . S ; A, $],
[A → . E, $],
[A → . id := E, $],
[S → . A, ;],
[S → . S ; A, ;],
[E → . E + id, $],
[E → . id, $],
[A → . E, ; ],
[A → . id := E, ; ],
[E → . E + id, +],
[E → . id, +],
[E → . E + id, ;],
[E → . id, ;]
```

FIGURE 2.5: CLOSURE I_0

2.5.3 First Transitions

We now have the following transitions :

- $I_0 \xrightarrow{S} I_1$
- $I_0 \xrightarrow{A} I_2$
- $I_0 \xrightarrow{E} I_3$
- $I_0 \xrightarrow{id} I_4$

Where,

I_1 = CLOSURE{ [S'→ S•, \$], [S → S• ; A, \$], [S → S• ; A, ;] }

I_2 = CLOSURE{ [S→ A•, \$], [S → A• , ;]}

I_3 = CLOSURE
{ [A→ E•, \$], [A → E•, ;], [E → E • +id , \$], [E → E • +id, ;], [E → E • +id, +] }

We perform this above process on each of I_1, I_2, I_3, I_4 recursively.

For example, GOTO(I_1, ;) = I_5 = CLOSURE{ [S→ S;•A, \$], [S → S;•A , ;] } and so on.

2.5.4 Set of all transitions in the Automaton

The set of all transitions of the Automata and it's implications with respect to Figure 2.6.

1. • $I_0 \xrightarrow{S} I_1$ GOTO[I_0,S] = I_1
   • $I_0 \xrightarrow{A} I_2$ GOTO[I_0,A] = I_2
   • $I_0 \xrightarrow{E} I_3$ GOTO[I_0,E] = I_3
   • $I_0 \xrightarrow{id} I_4$ ACTION[I_0,id] = shift I_1

2. • $I_1 \xrightarrow{;} I_4$ ACTION[I_1, ;] = shift I_5

3. • $I_3 \xrightarrow{+} I_6$ ACTION[I_3, +] = shift I_6

4. • $I_4 \xrightarrow{:=} I_7$ ACTION[I_4, :=] = shift I_7

5. • $I_5 \xrightarrow{A} I_8$ GOTO[I_0,S] = I_1
   • $I_5 \xrightarrow{E} I_3$ GOTO[I_0,S] = I_1
   • $I_5 \xrightarrow{id} I_4$ ACTION[I_5,id] = shift I_4

6. • $I_6 \xrightarrow{id} I_9$ ACTION[I_6,id] = shift I_9

7. • $I_7 \xrightarrow{E} I_{10}$ GOTO[I_7,E] = I_{10}
   • $I_7 \xrightarrow{id} I_{11}$ ACTION[I_7,id] = shift I_{11}

8. • $I_{10} \xrightarrow{+} I_6$ ACTION[I_{10},+] = shift I_6

2.5.5 States of the Automaton

On calculation of the CLOSURE of every state of the Automaton, we obtain the following results:

```
I0 = { [S' → . S, $],
       [S → . A, $],       [S → . A, ;],        [S → . S ; A, $],   [S → . S ; A, ;],
       [A → . id := E, $], [A → . id := E, ; ], [A → . E, $],       [A → . E, ; ],
       [E → . E + id, $],  [E → . E + id, +],   [E → . E + id, ;],
       [E → . id, $],      [E → . id, ;],       [E → . id, +] }

I1 = { [S'→ S ., $], [S → S . ; A, $], [S → S . ; A, ;] }
I2 = { [S → A ., $], [S → A ., ;] }
I3 = { [A → E ., $], [A → E ., ;], [E → E . + id, $], E → E . + id, ;], E → E . + id, +] }
I4 = { [A → id . := E, $], [A → id . := E, ;], [E → id ., $], [E→ id ., ;], [E → id ., +] }

I5 = { [S→S ; . A, ;], [S→S ; . A, $],
       [A→. id := E, $], [A→. id := E, ;], [A→. E, $], [A→. E, ;],
       [E→. E + id, $], [E→. E + id, ;], [E→. E + id, +], [E→. id, $], [E→. id, ;], [E→. id, +] }
I6 = { [E → E + . id, $], [E → E + . id, ;], [E → E + . id, +] }
I7 = { [A→id := . E, $], [A→id := . E, ;],
       [E→. E + id, $], [E→. E + id, ;], [E→. E + id, +],
       [E→. id, $], [E→. id, ;], [E→. id, +] }
I8 = { [S → S ;  A ., $], [S → S ;  A ., ;] }
I9 = { [E→ E + id ., $], [E→ E + id ., ;], [E→ E + id ., +]} )
I10= { [E→ E.+id, $], [E→ E.+id, ;], [E→ E.+id, +], [A→ id:=E., $], [A→ id:=E., ;]}
I11= { [E→ id ., $],  [E→ id ., ;],  [E→ id ., +]}
```

FIGURE 2.6: All the Final States of the Automaton

And, the states imply the following entries:

```
I0: none.

I1:
   [S'→ S ., $]   Action[I1,$] = accept
I2:
   [S → A ., $]   Action[I2,$] = reduce 3
   [S → A ., ;]   Action[I2,;] = reduce 3
I3:
   [A → E ., $]   Action[I3,$] = reduce 4
   [A → E ., ;]   Action[I3,;] = reduce 4
I4:
   [E → id ., $]  Action[I4,$] = reduce 7
   [E → id ., ;]  Action[I4,;] = reduce 7
   [E → id ., +]  Action[I4,+] = reduce 7

I5: none
```

```
I8:
   [S → S ;  A ., $]  Action[I8,$] = reduce 2
   [S → S ;  A ., ;]  Action[I8,;] = reduce 2

I9:
   [E→ E + id ., $]  Action[I9,$] = reduce 6
   [E→ E + id ., ;]  Action[I9,;] = reduce 6
   [E→ E + id ., +]  Action[I9,+] = reduce 6

I10:
   [A→ id:=E., $]    Action[I10,$] = reduce 5
   [A→ id:=E., ;]    Action[I10,;] = reduce 5

I11:
   [E→ id ., $]     Action[I11,$] = reduce 7
   [E→ id ., ;]     Action[I11,;] = reduce 7
   [E→ id ., +]     Action[I11,+] = reduce 7
```

FIGURE 2.7: ACTION-GOTO Table entries

2.5.6 Parse Table

The CFG results in the following Parse Table:

State	Action					GoTo			
	Id	**;**	**+**	**:=**	**$**	S'	S	A	E
0	S I4						I1	I2	I3
1		S I5			acc				
2		R 3			R 3				
3		R 4	S I6		R 4				
4		R 7	R 7	S I7	R 7				
5	S I4							I8	I3
6	S I9								
7	S I11								I10
8		R 2			R 2				
9		R 6	R 6		R 6				
10		R 5	S I6		R 5				
11		R 7	R 7		R 7				

FIGURE 2.8: LR(1) Parse Table

2.6 Example: LR(1) Parsing

In the aforementioned example, we may be interested in knowing whether a given string belongs to the context free grammar or not. This is done by the LR(1) Parser. We, will examine how the parser does this.

Let us consider, for example, the string ‘‘`id ; id`’’. We will use the aforementioned Parsing technique to determine whether it belongs to the CFG or not.

2.6.1 Basic Outline

String : `id ; id`

CFG :

- Terminals { $, ; , id, := , +}
- Non-terminals {S', S, A, E}
- Start Symbol = S'
- Productions :
  0. S' → S $
  1. S → S ; A
  2. S → A
  3. A → E
  4. A → id := E
  5. E → E + id
  6. E → id

Let "/" represent the extent of parsing that the string has undergone at any given point in time.

So, here goes:

```
1.  / id ; id $                 Shift
2.  id / ; id $                 Reduce by 6
3.  E / ; id $                  Reduce by 3
4.  A / ; id $                  Reduce by 2
5.  S / ; id $                  Shift
6.  S ; / id $                  Shift
7.  S ; id / $                  Shift
8.  S ; E / $                   Reduce by 6
9.  S ; A / $                   Reduce by 3
10.  S / $                      Reduce by 1
11.  Accept
```

2.6.2 LR(1) Parsing Technique

The output of the Parser is displayed in Figure 2.9.

2.6.3 Parse Table

The CFG results in the following Parse Table:

Pass	Stack	Symbol	Input	Action
1)	0		id ';' id $	shift
2)	0 4	id	';' id $	reduce by E -> id
3)	0 3	E	';' id $	reduce by A -> E
4)	0 1	A	';' id $	reduce by S -> A
5)	0 2	S	';' id $	shift
6)	0 2 5	S ';'	id $	shift
7)	0 2 5 4	S ';' id	$	reduce by E -> id
8)	0 2 5 3	S ';' E	$	reduce by A -> E
9)	0 2 5 7	S ':' A	$	reduce by S -> S ';' A
10)	0 2	S	$	accept

FIGURE 2.9: LR(1) Parser output for `id ; id`

Now, refer to Figure 2.8

1. The stack is initialized to state [0]. It encounters an id. ACTION[0,id] = s4 in the parse table. So, the stack pushes 4. Now, the stack becomes [0 4]

2. Now, the counter of the stack is at 4. Stack is [0 4]. It encounters a "; ". [4][;] in parse table says r7. So, it pops 4 and pushes the LHS of Production 7 i.e. E. So, temporarily the stack is [0 E] and this refers to the goto section of [0][E] which is 3. So, the stack pops E and pushes 3. The stack now becomes [0 3].

3. Now, the counter is at 3. And, the parser encounters ";" again in the string, since it hasn't been shifted. Now, [3][;] corresponds to r4. So, it reduces by production 4. Stack temporarily becomes [0 A] and then finally becomes [0 1] with the counter at 1.

4. It again encounters ";" in the string, and the parse table tells it to refer r3. Stack undergoes the transformations [0 1] -¿ [0 S] -¿ [0 2]

5. Counter at 2 now encounters ";" and performs s5. Stack becomes [0 2 5]. Parser advances through the string.

6. Stack counter at 5 now encounters id in the string. Parse table entry for [5][id] is s4. So, Stack becomes [0 2 5 4].

7. Stack counter at 4 now encounters "$" in the string. Parse table entry for [4][$] is r7. it reduces by 7. Stack eventually becomes [0 2 5 3]

8. There is another r4 and r1 performed by the stack, which eventually ends up becoming [0 2].

9. The Parse Table entry for stack counter at 2 and string value $ i.e. [2][$] is Accept. So, the LR(1) Parser accepts the string.

So, this is the process that a LR(1) Parser implements in checking strings with respect to CFGs.

.

Bibliography

1. John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Automata Theory, Languages and Computation*, 2011

2. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, 1986

3. Stephen M. Watt, University of Western Ontario, *Compiler Theory : Lecture Notes*, 2007

4. Laurence Rauchwerger, Texas A&M university, *Compiler Design : Lecture Notes*, 2006