LR Parsing

We first understand **Context Free Grammars**. Consider the input string: x+2*y

When scanned by a scanner, it produces the following stream of tokens:

id(x)
+
num(2)
*
id(y)

The goal is to parse this expression and construct a data structure (a parse tree) to represent it. A possible syntax for expressions is given by the following grammar G1:

```
E ::= E + T
   | E - T
   | T
T ::= T * F
   | T / F
   | F
F ::= num
   | id
```

where E, T and F stand for expression, term, and factor respectively. For example, the rule for E indicates that an expression E can take one of the following 3 forms: an expression followed by the token + followed by a term, or an expression followed by the token - followed by a term, or simply a term. The first rule for E is actually a shorthand of 3 productions:

E ::= E + T
E ::= E - T
E ::= T

G1 is an example of a context-free grammar; the symbols E, T and F are non-terminals and should be defined using production rules, while +, -, *, /, num, and id are terminals (i.e., tokens) produced by the scanner. The non-terminal E is the start symbol of the grammar.

In general, a context-free grammar (CFG) has a finite set of terminals (tokens), a finite set of non-terminals from which one is the start symbol, and a finite set of productions of the form:

A ::= X_1 X_2 ... X_n

where A is a non-terminal and each X_i is either a terminal or non-terminal symbol.

Given two sequences of symbols a and b (can be any combination of terminals and non-terminals) and a production A : : = $X_1X_2...X_n$, the form aAb = > a$X_1X_2...X_n$b is called a derivation. That is, the non-terminal symbol A is replaced by the rhs (right-hand-side) of the production for A. For example,

T / F + 1 - x => T * F / F + 1 - x

is a derivation since we used the production T := T * F.

Top-down parsing starts from the start symbol of the grammar S and applies derivations until the entire input string is derived (ie, a sequence of terminals that matches the input tokens). For example,

```
E => E + T
  => E + T * F
  => T + T * F
  => T + F * F
  => T + num * F
  => F + num * F
  => id + num * F
  => id + num * id
```

which matches the input sequence id(x) + num(2) * id(y). Top down parsing occasionally requires backtracking. For example, suppose that we used the derivation E => E - T instead of the first derivation. Then, later we would have to backtrack because the derived symbols will not match the input tokens. This is true for all non-terminals that have more than one production since it indicates that there is a choice of which production to use. We will learn how to construct parsers for many types of CFGs that never backtrack. These parsers are based on a method called predictive parsing. One issue to consider is which non-terminal to replace when there is a choice. For example, in T + F * F we have 3 choices: we can use a derivation for T, for the first F, or for the second F. When we always replace the leftmost non-terminal, it is called leftmost derivation.

In contrast to top-down parsing, bottom-up parsing starts from the input string and uses derivations in the opposite directions (i.e., by replacing the rhs sequence $X_1X_2...X_n$ of a production A : : = $X_1X_2...X_n$ with the non-terminal A. It stops when it derives the start symbol. For example,

```
   id(x) + num(2) * id(y)
=> id(x) + num(2) * F
=> id(x) + F * F
=> id(x) + T * F
=> id(x) + T
=> F + T
=> T + T
=> E + T
=> E
```

An **LR Parser** is a type of bottom-up Parser for context-free grammars that is very commonly used by computer programming language compilers (and other associated tools). The technique that can be used to parse a large class of context-free grammars is called LR(*k*) parsing; the 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse and the 'k' for the number of input symbols of look-ahead that are used in making parsing decisions. When (k) is omitted, it is assumed to be 1. LR parsing is attractive for a variety of reasons.

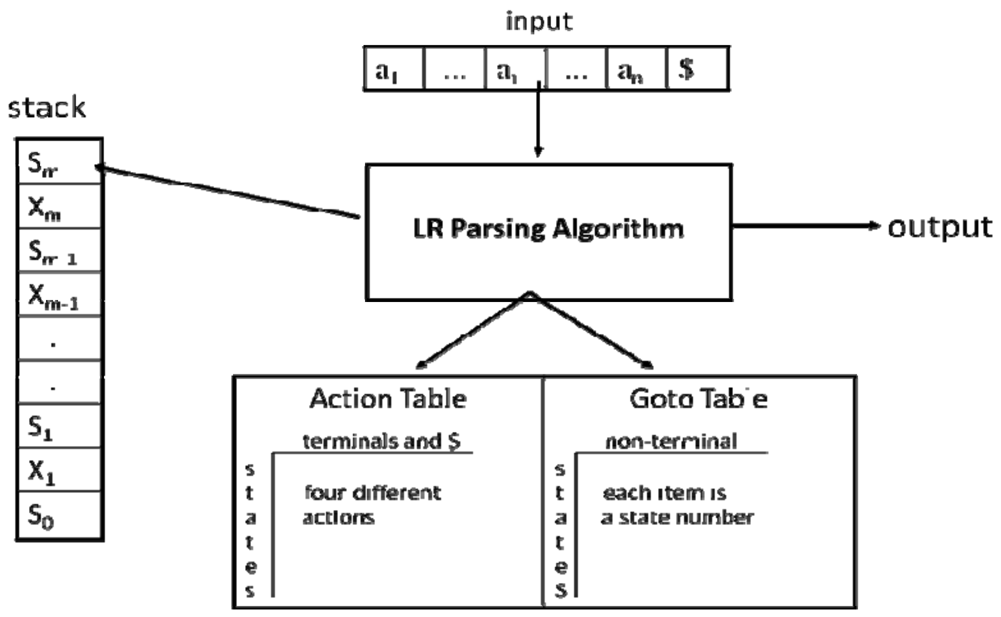
1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.

2. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.

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 parsers.

4. 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.

The principal drawback of the method is that it is too much work to construct an LR parser by hand for a typical programming language grammar. One needs a specialized tool – an LR parser generator. With such a generator, one can write a context-free grammar and have the generator automatically produce a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator can locate these constructs and inform the compiler designer of their presence.



There are three techniques for constructing an LR parsing table for a grammar.

- Simple LR (SLR for *short*), is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammars on which the other methods succeed. A Simple LR parser or SLR parser is an LR parser for which the parsing tables are generated as for an LR(0) parser except that it only performs a reduction with a grammar rule A → w if the next symbol on the input stream is in the follow set of A. Such a parser can prevent certain shift-reduce and reduce-reduce conflicts that occur in LR(0) parsers and it can therefore deal with more grammars. However, it still cannot parse all grammars that can be parsed by an LR(1) parser. A grammar that can be parsed by an SLR parser is called a SLR grammar.

  Example

  A grammar that can be parsed by an SLR parser [but not by an LR(0) parser] is the following:

  (1) E → 1 E

  (2) E → 1

  Constructing the action and goto table as is done for LR(0) parsers would give the following item sets and tables:

  Item set 0

  S → · E
  + E → · 1 E
  + E → · 1

  Item set 1

  E → 1 · E
  E → 1 ·
  + E → · 1 E
  + E → · 1

  Item set 2
  S → E ·

  Item set 3

  E → 1 E ·

The action and goto tables:

	action		*goto*
state	**1**	**$**	**E**
0	s1		2
1	s1/r2	r2	3
2		acc	
3	r1	r1	

As can be observed there is a shift-reduce conflict for state 1 and terminal '1'. However, the follow set of E is { $ } so the reduce actions r1 and r2 are only valid in the column for $. The results are the following conflict-less action and goto table:

	action		*goto*
state	**1**	**$**	**E**
0	s1		2
1	s1	r2	3
2		acc	
3		r1	

- Canonical LR, is the most powerful and expensive. A canonical LR parser or LR(1) parser is an LR parser whose parsing tables are constructed in a similar way as with LR(0) parsers except that the items in the item sets also contain a follow, i.e., a terminal that is expected by the parser after the right-hand side of the rule. Such an item for a rule A -> BC is for example of the form

  A -> B · C, a

  which would mean that the parser has read a string corresponding to B and expects next a string corresponding to C followed by the terminal 'a'. LR(1) parsers can deal with a very large class of grammars but their parsing tables are often very big. This can often be solved by merging item sets if they are identical except for the follows, which results in so-called LALR parsers.

- Look Ahead LR (LALR for *short*), is intermediate in power and cost between the other two. This method is often used in practice because the tables obtained by it are considerably smaller than the Canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar.

References

[1] http://www.economicexpert.com/a/Canonical:LR:parser.html assessed on December 13, 2009 at 2235 Hrs
[2] http://www.economicexpert.com/a/Simple:LR:parser.htm assessed on December 13, 2009 at 2305 Hrs
[3] http://lambda.uta.edu/cse5317/notes/node12.html assessed on December 13, 2009 at 2330 Hrs
[4] http://203.208.166.84/dtanvirahmed/cse309N/CSE309-4-Rest.ppt retrieved on December 11*, 2009 at 1620 Hrs*