Compiler Design Project
Phase#2

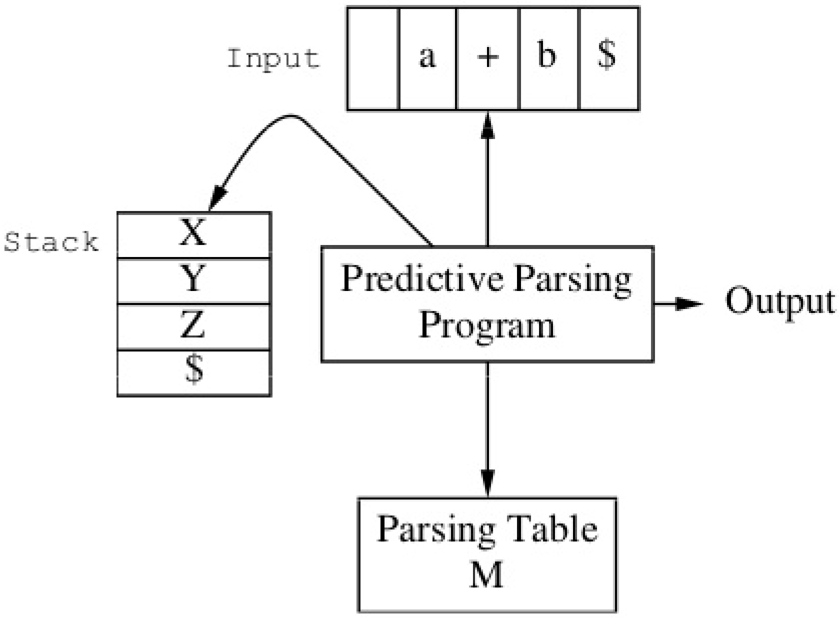
Parser Generator

Abobakr Abdelaziz ID : 2
Ahmed Ali Elsayed ID : 8
Elsayed Akram Elsayed ID : 16
Fares Medhat El-Saadawy ID: 47

PROF. Nagia Ghanem

Overview

A top-down parser that uses a one-token lookahead is called an **LL(1)** parser.
The first **L** indicates that the input is read from left to right.
The second **L** says that it produces a left-to-right derivation.
And the **1** says that it uses one lookahead token.
The parser needs to find a production to use for nonterminal N when it sees lookahead token t.
To select which production to use, it suffices to have a table that has, as a key, a pair (N, t) and gives the number of a production to use.

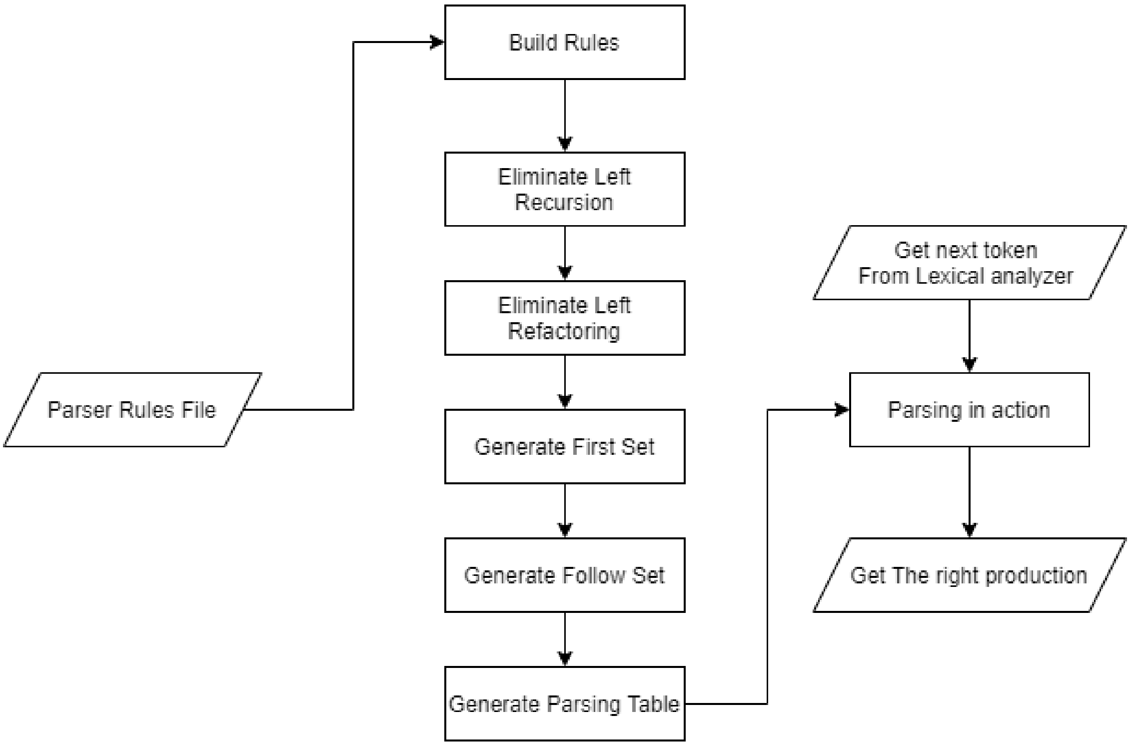


Img from : [Top-Down parsing (slideshare.net)]

How to run

```
cd project_directory
run lex_rules.txt parser_rules.txt lab_program.txt
```

Program Flow



Algorithms

Algorithms for parsing the input file:

Read each line and parse it to get the left hand side and right hand side of a production.

```
parseRule:
    if the line starts with ‘|’:
        Parse the right hand side and add it to the last production
    Parse the left hand side before the ‘=’ sign
    Parse the right hand side after the ‘=’ sign and at each ‘|’ create
new vector to push it to the right hand side vector of vectors
```

Eliminate left-recursion:

A grammar cannot be immediately left-recursive, but it still can be left-recursive. By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

```
for i from 1 to n do {
   for j from 1 to i-1 do {
      replace each production
      Ai -> Aj y by Ai -> a1 y | .... | ak y
      where Aj -> a1 | ... | ak
   }
   eliminate immediate left-recursions among Ai productions
}
```

Eliminate left-refactoring:

A predictive parser (a top-down parser without backtracking) insists that the grammar must be left-factored.
Grammar -> a new equivalent grammar suitable for predictive parsing.

For example :
let grammar is : A -> xB | xC | yD | yE | F
Should turned into
A -> xG | yH | F
G -> B | C
H -> C | D

```
For i from 0 to rules.size() {
  Get indices for vectors with the same first Element
  Get the number of longest common prefix between them
  Push this common elements to vector and new non lhs will
  Pushed back to this vector
  This vector will be pushed to rhs
  New rhs will have the vectors of elements without common prefix
  Remove the vectors with the same prefix from rhs
  set rhs now
  Make new production with new non terminal and new rhs
  This new production will pushed to vector of rules
}
```

Generate First Set:

FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X.
Rules to construct the first set :

```
If X is a terminal symbol FIRST(X)={X}

If X is a non-terminal symbol and X -> E is a production rule then
E is in FIRST(X)

If X is a non-terminal symbol and X -> Y1 Y2..Yn is a production
rule and if a terminal t in FIRST(Yi) and E is in all FIRST(Yj) for
j=1,...,i-1 then t is in FIRST(X).

if E is in all FIRST(Yj) for j=1,...,n then E is in FIRST(X).
```

Generate Follow Set:

FOLLOW(X) for a grammar symbol X is the set of the terminals which occur immediately after (follow) the non-terminal X in the strings derived from the starting symbol.

Rules to construct the follow set :

```
If $ is the start symbol $ is in FOLLOW(S)

If A -> xBy is a production rule >> everything in FIRST(y) is
FOLLOW(B) except E

If ( A -> xB is a production rule ) or ( A -> xBy is a production
rule and E is in FIRST(y)) >> everything in FOLLOW(A) is in
FOLLOW(B).
```

Generate parsing table from follow and first :

What is the Parsing Table?

- A two-dimensional array M[A,a]
- Each row is a non-terminal symbol
- Each column is a terminal symbol or the special symbol $
- Each entry holds a production rule.

Rules to build the parsing table :

```
for each production rule A -> x of a grammar G :
  for each terminal a in FIRST(x) add A -> x to M[A,a]

  If E in FIRST(x) for each terminal a in FOLLOW(A) add A -> x to
M[A,a]

  If E in FIRST(x) and $ in FOLLOW(A)  add A -> x to M[A,$]

  If the First(x) not contains E then for each terminal t in
follow(A) : if M[A,t] = SYNC if not empty
```

Parsing Actions:

The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
There are four possible parser actions.

1) If X and a are $ parser halts (successful completion)
2) If X and a are the same terminal symbol (different from $) parser pops X from the stack, and moves the next symbol in the input buffer.
3) If X is a non-terminal parser look at the parsing table entry M[X,a]. If M[X,a] holds a production rule XY1Y2...Yk, it pops X from the stack and pushes Yk,Yk-1,...,Y1 into the stack. The parser also outputs the production rule XY1Y2 ...Yk to represent a step of the derivation.

4) If none of the above error
   - All empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

Error recovery : panic mode error recovery

For each nonterminal A, mark the entries M[A,a] as synch if a is in the follow set of A. So, for an empty entry, the input symbol is discarded. This should continue until either:

1) an entry with a production is encountered. In this case, parsing continues as usual.

2) an entry marked as synch is encountered. In this case, the parser will pop that non-terminal A from the stack. The parsing continues from that state.

Data Structures:

```
Class Elem :
   string id

Class NonTerminal : Elem
Class Terminal : Elem

Class Production :
   NonTerminal* lhs
   vector<vector<Elem*>> rhs

Class LLParser :
   hashMap<NonTerminal*, hashMap<Terminal*, vector<Elem*>>> parsingTable;
   stack<Elem*> LLStack;
   vector<Terminal*> outputTerminals;
   hashMap<string, Terminal*> terminalsMapping;

Class LLParserGenerator:
   vector<Production*> rules;
   hashMap<NonTerminal*, unordered_set<Terminal*>> first, follow;
   hashMap<NonTerminal*, bool> isFirstBuild,isFollowBuild;
   hashMap<NonTerminal*, Production*> rulesMapping;
   NonTerminal* startState;

Class CFGBuilder :
   fstream rulesFile;
   vector<Production*> procList;
   hashMap<string, Production*> rulesMapping;
   hashMap<string, Terminal*> terminalsMapping;
   hashMap<string, NonTerminal*> nonTerminalsMapping;
```

Output for the Given Test Program & Parsing Table

[Parsing Table]()

```
int x;
x = 5;
if (x > 2)
{
     x = 0;
}
```

```
[redacted]run lex_rules.txt parser_rules.txt lab_program.txt
METHOD_BODY $
STATEMENT_LIST $
STATEMENT STATEMENT_LIST# $
DECLARATION STATEMENT_LIST# $
PRIMITIVE_TYPE id ; STATEMENT_LIST# $
int id ; STATEMENT_LIST# $
int id ; STATEMENT STATEMENT_LIST# $
int id ; ASSIGNMENT STATEMENT_LIST# $
int id ; id assign EXPRESSION ; STATEMENT_LIST# $
int id ; id assign SIMPLE_EXPRESSION EXPRESSION? ; STATEMENT_LIST# $
int id ; id assign TERM SIMPLE_EXPRESSION# EXPRESSION? ; STATEMENT_LIST# $
int id ; id assign FACTOR TERM# SIMPLE_EXPRESSION# EXPRESSION? ; STATEMENT_LIST# $
int id ; id assign num TERM# SIMPLE_EXPRESSION# EXPRESSION? ; STATEMENT_LIST# $
int id ; id assign num SIMPLE_EXPRESSION# EXPRESSION? ; STATEMENT_LIST# $
int id ; id assign num EXPRESSION? ; STATEMENT_LIST# $
int id ; id assign num ; STATEMENT_LIST# $
int id ; id assign num ; STATEMENT STATEMENT_LIST# $
int id ; id assign num ; IF STATEMENT_LIST# $
int id ; id assign num ; if ( EXPRESSION ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( SIMPLE_EXPRESSION EXPRESSION? ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( TERM SIMPLE_EXPRESSION# EXPRESSION? ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( FACTOR TERM# SIMPLE_EXPRESSION# EXPRESSION? ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id TERM# SIMPLE_EXPRESSION# EXPRESSION? ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id SIMPLE_EXPRESSION# EXPRESSION? ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id EXPRESSION? ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id relop SIMPLE_EXPRESSION ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id relop TERM SIMPLE_EXPRESSION# ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id relop FACTOR TERM# SIMPLE_EXPRESSION# ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num TERM# SIMPLE_EXPRESSION# ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num SIMPLE_EXPRESSION# ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { IF? STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { ASSIGNMENT } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign EXPRESSION ; } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign SIMPLE_EXPRESSION EXPRESSION? ; } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign TERM SIMPLE_EXPRESSION# EXPRESSION? ; } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign FACTOR TERM# SIMPLE_EXPRESSION# EXPRESSION? ; } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign num TERM# SIMPLE_EXPRESSION# EXPRESSION? ; } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign num SIMPLE_EXPRESSION# EXPRESSION? ; } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign num EXPRESSION? ; } else { STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign num ; } STATEMENT } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign num ; } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign num ; } $
accept
```

Recovery Example Program

Prog:

```
int x; ;;
x = 5;
if (x > 2)
{
      x = 0;
} else {
      else {
```

```
    }
}
```

The extra semi columns output:

```
METHOD_BODY $
STATEMENT_LIST $
STATEMENT STATEMENT_LIST# $
DECLARATION STATEMENT_LIST# $
PRIMITIVE_TYPE id ; STATEMENT_LIST# $
STATEMENT_LIST# -;-> Nothing in table
Discard this terminal.
STATEMENT_LIST# -;-> Nothing in table
Discard this terminal.
int id ; STATEMENT_LIST# $
```

The extra else output:

```
int id ; id assign num ; if ( id relop num ) { id assign num EXPRESSION? ; } else { STATEMENT } STATEMENT_LIST# $
STATEMENT -else-> Nothing in table
Discard this terminal.
STATEMENT -{-> Nothing in table
Discard this terminal.
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { STATEMENT } STATEMENT_LIST# $
STATEMENT_LIST# -}-> Nothing in table
Discard this terminal.
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { } STATEMENT_LIST# $
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { } $
accept
```

Assumptions & their Justifications

If the terminal t exists in the parser rules but not in the lexical rules, it's automatically added into the set of terminals.