Top Down Parsers

- Recursive Descent Parsers
  - Can be hand written (atl/0, Pascal-p4)
  - Can be "generated"

- Non-recursive predictive parsing
  - Table based PDA
  - Usually generated tables
  - Table: $X \times a \rightarrow P$
  - $f(X,a) = P$ (P is a production)
  - Stack has what we expect to see in the reverse order.

- Top Down typically finds a leftmost derivation
Predictive Parsing Algorithm (LL)

Stack <-- G$  # G is goal non-terminal
Input <-- w$
a <-- first token of w
repeat
    X <-- top of stack
    if X is a terminal
        if X = a
            pop X, a <-- next token, possible semantic action
        else
            error
    else
        if M[X,a] = X -> Y1 ... Yk then
            pop X, push Yk Yk-1 ... Y1
            output X -> Y1 ... Yk, possible semantic action
        else
            error
    until X = $ (Stack is empty!)
If a is not EOF, error
Bottom up parsers (Shift reduce)

- Build tree from the bottom up
- Start with the leaves
- LR, operator Precedence
- Typically 2 tables
- Partial productions are on stack
- Finds a rightmost derivation (in reverse)
void shift_reduce_driver (void)
{
    push (S0);
    T = scanner();
    while (TRUE) {
        S = top of stack;
        switch (action[S][T]) {
            case ERROR:
                handle_error();
                break;
            case ACCEPT:
                clean_up_and_finish();
                return;
            case SHIFT:
                push (go_to[S][T]);
                T = scanner();
                break;
        }
    }
}
case REDUCE:
    i = production number for X -> X1, X2, ... Xn;
    pop n symbols;
    S1 = top of stack;
    push (go_to[S1][X]);
    break;

}
Both methods need to analyze grammars

Parser generators need to read grammars:
- □ terminal
- □ non-terminals (variables)
- □ start symbol
- □ productions
  - □ left hand side
  - □ length of right hand side
  - □ symbols on right hand side
Nullable non-terminals

A is nullable iff $A \Rightarrow^+ \lambda$

1) mark all A such that $A \Rightarrow \lambda$

2) mark all B such that $B \Rightarrow C_1 \ldots C_n$
   and $C_1 \ldots C_n$ are marked nullable

3) repeat 2 until until no more Bs can be marked

$S \Rightarrow A\ B | C$
$A \Rightarrow a\ A | a$
$B \Rightarrow b\ B | \lambda$
$C \Rightarrow c\ C | \lambda$
Follow and First sets

Follow(A) = set of terminal symbols that may follow A in some sentential form.

Follow(A) = { a in Terminals ∣ S =>+ alpha A a beta }
union ( if S =>+ alpha A then { lambda } else { } )

First (alpha) = { a in Terminals ∣ alpha =>* a beta }
union ( if alpha =>* lambda then { lambda } else { } )

First is set of first terminals in some sentential form
(often applied to variables ...)

if alpha = a beta, First(alpha) = { a }
if alpha = A beta, First(alpha) =
  First(A) union ( if A is nullable First(beta) else { } )
Computation of first and follow sets.

First (alpha)
let alpha = X1 ... Xn

if n = 0, return {lambda}
result <-- first[X1] - {lambda}
for ( i = 2; i<= n; i++)
  if lambda in first[Xi-1]
    result <-- result union (first[Xi] - {lambda})
  else
    break

if (i == n+1 & lambda in first[Xn])
  result <-- result union {lambda}

return result
first[X] sets?

for all a in terminals set first[a] <-- {a}

for all A in variables
  if A -> lambda is a production
    first[A] <-- {lambda}
  else
    first[A] <-- {}

for all productions of the form A -> a beta
  first[A] <-- first[A] union {a}

do
  changes <-- false
  for all productions of the form A -> B beta
    first[A] <-- first[A] union First(B beta)
    if first[A] has changed, changes <-- true
  until no changes
Sample Grammar

p -> BEGIN stmts END
stmts -> stmt ";" stmts
stmts ->
stmt -> SSTMT
stmt -> BEGIN stmts END

first sets:

BEGIN: BEGIN
END: END
SSTMT: SSTMT
;: ;
p: BEGIN
stmts: Lambda SSTMT BEGIN ;
stmt: SSTMT BEGIN
Follow Set Algorithm

a) for A in Variables  follow[A] <-- { }

b) follow[S] <-- {Lambda}

c) do
   changes <-- false
   for each production of the form A -> alpha B beta
      follow[B] <-- follow[B] union (First(beta) - {Lambda})
   if (Lambda in First(beta)) then
      follow[B] <-- follow[B] union follow[A]
   if (follow[B] has changed) 
      changes <-- true
   end for
until no changes
Another example

1) S -> a S z
2) S -> A
3) A -> b A y
4) A -> B
5) A -> Lambda
6) B -> c B x
7) B -> m

first sets:
S:  a b Lambda c m
A:  b Lambda c m
B:  c m

follow sets:
S:  z Lambda
A:  y z Lambda
B:  x y z Lambda
Parser Generators

Top Down

- Given a "lookahead" token, predict the rule to push.
- Predict function ... know the difference between
  - A -> X1 .... Xn
  - A -> Y1 .... Yn

Predict ( A -> X1 ... Xn ) =
  if lambda in First(X1 ... Xn) then
    (First(X1 ... Xn) - {lambda}) union follow[A]
  else
    First(X1 ... Xn)
Predict and Parse Table

1) $S \rightarrow a \ S \ z$
   \[a\]

2) $S \rightarrow A$
   \[b \ c \ m \ z\]

3) $A \rightarrow b \ A \ y$
   \[b\]

4) $A \rightarrow B$
   \[c \ m\]

5) $A \rightarrow \text{Lambda}$
   \[y \ z\]

6) $B \rightarrow c \ B \ x$
   \[c\]

7) $B \rightarrow m$
   \[m\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>m</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>