Compiler Construction

Lecture 5 - Top-Down Parsing

What is top-down parsing?

- Top-down parsing is a parsing-method where a sentence is parsed starting from the root of the parse tree (with the *"Start"* symbol), working recursively down to the leaves of the tree (with the terminals).
- In practice, top-down parsing algorithms are easier to understand than bottom-up algorithms.
- Not all grammars can be parsed top-down, but most context-free grammars can be parsed bottom-up.

An example of top-down parsing

Let's consider the expression grammar:

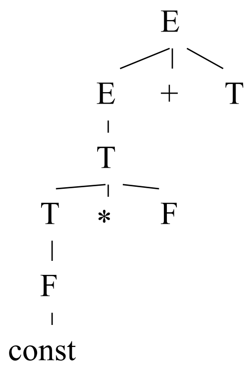
E ::= E + T | T

T ::= T * F | F

F ::= **id** | **const** | **(** E **)**

How will it begin parsing the expression:

3*x + y* z



LL(k) grammars

- Top-down grammars are referred to as LL(k) grammars:
  - The first L indicates ***L***eft-to-Right scanning.
  - The second L indicates ***L***eft-most derivation
  - The k indicates k lookahead characters.
- We will be examining LL(1) grammars, which spot errors at the earliest opportunity but provide strict requirements on our grammars.

LL(1) grammars

- LL(1) grammars determine from a single lookahead token which alternative derivation to use in parsing a sentence.
- This requires that if a nonterminal A has two different productions:

  A ::= α   and   A ::= β

  - that a and ß cannot begin with the same token.
  - α ***or*** ß can derive an empty string but not both.
  - if ß $=>^*$ ε, α cannot derive any string that begins with a token that could immediately follow A.

LL(1) grammars (continued)

If you look at the first token of expression

3*x + y*z

(which is **const**) and the productions for the start symbol E

E ::= E + T | T

How can you tell whether it derives ***E*** + ***T*** or simply ***T***? This requires information about the subsequent tokens.

LL(1) grammars (continued)

It becomes necessary to convert many grammars into LL(1). The most common conversions involve:

- Removing left-recursion (whether it is direct or indirect)
- Factoring out any terminals found out the beginning of more than one production for a given nonterminal

Removing left-recursion

- Aho, Sethi and Ullman show that left recursion of the form:

$$A ::= A\alpha \mid \beta$$

can be converted to right-recursion (which ***<u>is</u>*** LL(1)) by replacing it with the following productions:

$$A ::= \beta A'$$

$$A' ::= \alpha A' \mid \varepsilon$$

Removing indirect left recursion

Indirect recursion has nonterminals appear on the right-hand side of productions which appear in its own right sentential form: e.g., $A ::= B\ \alpha \mid c$

$B ::= B\ \beta \mid A\ \delta \mid d$

This can be removed by arranging the nonterminals in order and by then substituting for A in B’s right sentential form:

$A ::= B\ \alpha \mid c$

$B ::= B\ \beta \mid B\ \alpha\ \delta \mid c\ \delta \mid d$

Now we simply eliminate the direct left-recursion:

$A ::= B\ \alpha \mid c$

$B ::= c\ \delta\ B' \mid d\ B'$

$B' ::= \beta B' \mid \alpha\ \delta B' \mid \varepsilon$

Left-Factoring

Many grammars have the same prefix symbols at the beginning of alternative right sentential forms for a nonterminal:

e.g., $A ::= \alpha\ \beta \mid \alpha\ \gamma$

We replace these production with the following:

$A ::= \alpha\ A'$

$A' ::= \beta \mid \gamma$

Converting an expression grammar into LL(1) form

- Our expression grammar is:

$$E ::= E + T \mid T$$
$$T ::= T * F \mid F$$
$$F ::= \textbf{id} \mid \textbf{const} \mid \textbf{(}\ E\ \textbf{)}$$

- Following our rule for removing direct left-recursion, our grammar becomes:

$$E ::= T\,E'$$
$$E' ::= + T\,E' \mid \varepsilon$$
$$T ::= F\,T'$$
$$T' ::= * F\,T' \mid \varepsilon$$
$$F ::= \textbf{id} \mid \textbf{const} \mid \textbf{(}\ E\ \textbf{)}$$

Parse Table

Once the grammar is in LL(1) form, we create a table showing which production we use in parsing each nonterminal for every possible lookahead token:

	E	E’	T	T’	F
+		2		6	
*				5	
(	1		4		9
)		3		6	
id	1		4		7
const	1		4		8
$		3		6	

1	E ::= TE’
2	E’ ::= +TE’
3	E’ ::= ε
4	T ::= FT’
5	T’ ::= *FT’
6	T’ ::= ε
7	F ::= id
8	F ::= const
9	F ::= (E)

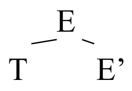
Parsing an expression using the LL(1) parse table

Let's take a look at the expression

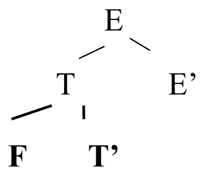
3*x + y

Our parse tree is initially just the start symbol *E* and our lookahead token is **const** (the lexeme is **3**)

The production for E and a lookahead token of **const** is is #1, making our parse tree

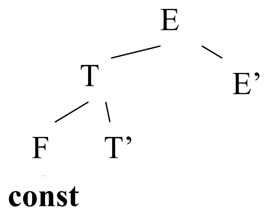


The production for T and a lookahead token of **const** is is #4, making our parse tree:



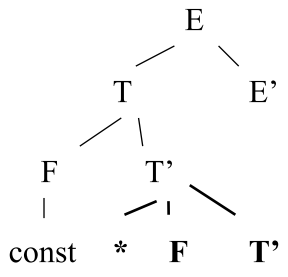
Parsing an expression using the LL(1) parse table (continued)

The production for F and a lookahead token of **const** is #8, making our parse tree:



Since we have now matched the token, we get a new lookahead

The production for T' and a lookahead token of * is is #5, making our parse tree:

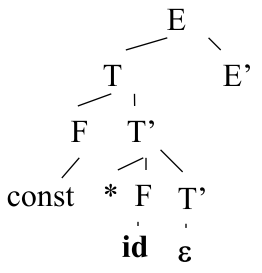


We get another lookahead

Parsing an expression using the LL(1) parse table (continued)

The production for F and a lookahead token of **id** is #7, making our parse tree:

We get a new lookahead

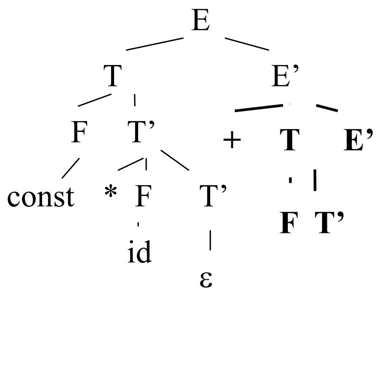


The production for T’ and a lookahead token of + is #6, making our parse tree:

Parsing an expression using the LL(1) parse table (continued)

The production for E’ and a lookahead token of + is #2, making our parse tree:

We get a new lookahead

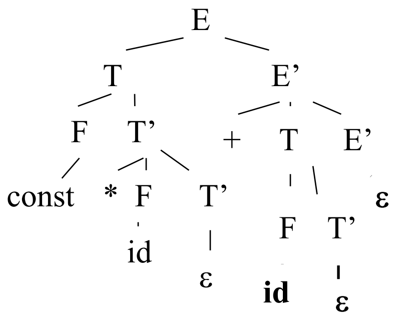


The production for T and a lookahead token of id is #4, making our parse tree:

Parsing an expression using the LL(1) parse table (continued)

The production for F and a lookahead token of **id** is #7, making our parse tree:

We get a new lookahead



Having reached the EOF (represented by **$**), the productions for T’ and E’ are 6 and 3 respectively. Our parse tree is complete.

FIRST and FOLLOW sets

- Since LL(1) grammars are ***predictive*** (the first lookahead determines how we parse the nonterminal), the set of tokens that can appear at the beginning of any derivation for that nonterminal becomes extremely important. This is the ***FIRST set***.
- Since LL(1) grammars include epsilon productions, the set of token that can appear immediately following the nonterminal becomes important in determining the FIRST set and is called the ***FOLLOW set***.

FIRST sets

- Definition: FIRST(A) (where A is a nonterminal) is the set of symbols beginning strings generated by A.
- Examples
  - FIRST(*E*) = { ***id***, ***const***, ***(*** }
  - FIRST (*E'*) = { +, ***)***, ***$*** }
  - FIRST(*T*) = {***id***, ***const***, ***(*** }
  - FIRST(*T'*) = {+, *, ***)***, ***$*** }
  - FIRST(F) = {***id***, ***const***, ***(*** }

FOLLOW sets

- Definition: FOLLOW(A) (where A is a nonterminal) is a set of symbol appearing to the right of A in a sentential form.
- Examples:
  - FOLLOW(E) = { ***)***, ***$*** }
  - FOLLOW(E') = {***)***, ***$*** }
  - FOLLOW(T) = (+, ***)***, ***$*** }
  - FOLLOW(T') = { +, ***)***, ***$*** }
  - FOLLOW(F) = { *, +, ***)***, ***$***}

Computing FIRST sets

To compute FIRST sets:

```
WHILE there are still terminals that may be added:
   FOR each production and for each nonterminal (in
   the form A ::= $\beta_i$:
         IF $\beta_1$ is a terminal, add it to the FIRST set for
                  A (& Production i)
         ELSE IF $\beta_1$ is $\varepsilon$, add the FOLLOW set for A
                  to the FIRST set for A (& Production i)
         ELSE ($\beta_1$ is a nonterminal), add the FIRST set
          for $\beta_1$ to the FIRST set for A (& Production i).
```

Computing FOLLOW sets

```
Initialize FOLLOW(S) to { $ }
Initialize the other FOLLOW sets to { }
WHILE there are terminals that can be added:
   FOR each nonterminal B:
         Look for productions $P_i$ in the form A ::= $\alpha$ B $\gamma$
         IF $\gamma$ is a terminal, add it to the set FOLLOW(B)
         ELSE IF $\gamma = \varepsilon$, add FOLLOW(A) to FOLLOW(B)
         ELSE ($\gamma$ is a nonterminal), add FIRST($\gamma$ ) to
                  FOLLOW(B)
```

Determining FIRST sets for LL(1) grammars

Let’s take another look at our expression grammar in LL(1) form:

1	E ::= TE’	FIRST(1) = FIRST(T)
2	E’ ::= +TE’	FIRST(2) = { + }
3	E’ ::= ε	FIRST(3) = FOLLOW(E’)
4	T ::= FT’	FIRST(4) = FIRST(F)
5	T’ ::= *FT’	FIRST(5) = { * }
6	T’ ::= ε	FIRST(6) = FOLLOW(T’)
7	F ::= id	FIRST(7) = { ***id*** }
8	F ::= const	FIRST(8) = { ***const*** }
9	F ::= (E)	FIRST(9) = { ***(*** }

It becomes useful to work with the FIRST(i), which is the first set for the derivations of the nonterminal coming from production i.

Determining FIRST sets for LL(1) grammars (continued)

FIRST(F) = FIRST(7) ∪ FIRST(8) ∪ FIRST (9)
= { ***id***, ***const***, ***(*** }

Therefore, FIRST(4) = FIRST (F) = { ***id***, ***const***, ***(*** }

Also, FIRST(T) = FIRST (4) = { ***id***, ***const***, ***(*** }

1	E ::= TE’	FIRST(1) = { ***id***, ***const***, ***(*** }
2	E’ ::= +TE’	FIRST(2) = { + }
3	E’ ::= ε	FIRST(3) = FOLLOW(E’)
4	T ::= FT’	FIRST(4) = { ***id***, ***const***, ***(*** }
5	T’ ::= *FT’	FIRST(5) = { * }
6	T’ ::= ε	FIRST(6) = FOLLOW(T’)
7	F ::= id	FIRST(7) = { ***id*** }
8	F ::= const	FIRST(8) = { ***const*** }
9	F ::= (E)	FIRST(9) = { ***(*** }

Determining FIRST sets for LL(1) grammars (continued)

FOLLOW(E) is initially { ***$*** }.

Since E is on the right side of Production #9, we add ***)*** to the set.

Since E’ is in both Productions 1 and 2 without anything following it (and Production is recursive, adding nothing), we add FOLLOW(E) to FOLLOW(E’)

FOLLOW(E’) = FOLLOW(E) = { ***)*** , ***$*** }

Similarly,

FOLLOW(T’) = FOLLOW(T) = FIRST(E’) ∪ FOLLOW(E’)
= { + } ∪ { ***)*** , ***$*** }

Determining FIRST sets for LL(1) grammars (continued)

After determining the FOLLOW sets for E’ and T’, we have:

1	E ::= TE’	FIRST(1) = { ***id***, ***const***, ***(*** }
2	E’ ::= +TE’	FIRST(2) = { + }
3	E’ ::= ε	FIRST(3) = { ***)*** , ***$*** }
4	T ::= FT’	FIRST(4) = { ***id***, ***const***, ***(*** }
5	T’ ::= *FT’	FIRST(5) = { * }
6	T’ ::= ε	FIRST(6) = { + , ***)*** , ***$*** }
7	F ::= id	FIRST(7) = { ***id*** }
8	F ::= const	FIRST(8) = { ***const*** }
9	F ::= (E)	FIRST(9) = { ***(*** }

How do we use this information in Parsing?

To determine which production to use in expanding a nonterminal.

Parse Table - the final result

By looking up the token and nonterminal, we determine how to expand a given nonterminal.

We can build the parse recursively, or nonrecursively by using a stack.

	E	E’	T	T’	F
+		2		6	
*				5	
(	1		4		9
)		3		6	
id	1		4		7
const	1		4		8
$		3		6	

1	E ::= TE’
2	E’ ::= +TE’
3	E’ ::= ε
4	T ::= FT’
5	T’ ::= *FT’
6	T’ ::= ε
7	F ::= id
8	F ::= const
9	F ::= (E)

The LL(1) JASON Grammar

1 *Program* ::= *Header DeclSec Block* **.**

2 *Header* ::= **program identifier ;**

3 *DeclSec* ::= *VarDecls ProcDecls*

4 *VarDecls* ::= *VarDecl VarDecls*

5 *VarDecls* ::= ε

6 *VarDecl* ::= *DataType IdList;*

7 *DataType* ::= **integer**

8 *DataType* ::= **real**

9 *IdList* ::= **identifier** *MoreIdList*

The LL(1) JASON Grammar (continued)

10 *MoreIdList* ::= **, identifier** *MoreIdList*
11 *MoreIdList* := ε
12 *ProcDecls* ::= *ProcDecl ProcDecls*
13 *ProcDecls* ::= ε
14 *ProcDecl* ::= *ProcHeader DeclSec Block* **;**
15 *ProcHeader* ::= **procedure identifier** *ParamList* **;**
16 *ParamList* ::= **(** *ParamDecls* **)**
17 *ParamList* ::= ε
18 *ParamDecls* ::= *ParamDecl MoreParamDecls*

The LL(1) JASON Grammar (continued)

19 *MoreParamDecls* ::= **;** *ParamDecl MoreParamDecls*
20 *MoreParamDecls* ::= ε
21 *ParamDecl* ::= *DataType* **identifier**
22 *Block* ::= **begin** *Statements* **end**
23 *Statements* ::= *Statement MoreStatements*
24 *MoreStatements* ::= **;** *Statement MoreStatements*
25 *MoreStatements* ::= ε
26 *Statement* ::= **read identifier**
27 *Statement* ::= **set identifier =** *Expression*

The LL(1) JASON Grammar (continued)

28 *Statement* ::= **write identifier**

29 *Statement* ::= **if** *Condition* **then** *Statements ElseClause* **endif**

30 *Statement* ::= **while** *Condition* **do** *Statements* **endwhile**

31 *Statement* ::= **until** *Condition* **do** *Statements* **enduntil**

32 *Statement* ::= **call identifier** *ArgList*

33 *Statement* ::= ε

34 *ElseClause ::=* **else** *Statements*

35 *ElseClause ::=* ε

36 *ArgList ::=* **(** *Args* **)**

The LL(1) JASON Grammar (continued)

37 *ArgList* ::= ε

38 *Args* ::= **identifier** *MoreArgs*

39 *MoreArgs* ::= , **identifier** *MoreArgs*

40 *MoreArgs* ::= ε

41 *Condition* ::= *Expression RelOp Expression*

42 *RelOp* ::= **=**

43 *RelOp ::=* **!**

44 *RelOp ::=* >

45 *RelOp ::=* <

The LL(1) JASON Grammar (continued)

46 *Expression ::= Term MoreExpression*
47 *MoreExpression ::= AddOp Term MoreExpression*
48 *MoreExpression ::=* ε
49 *Term* ::= *Factor MoreTerm*
50 *MoreTerm* ::= *MultOp Factor MoreTerm*
51 *MoreTerm* ::= ε
52 *Factor* ::= **identifier**
53 *Factor*::= **constant**
54 *AddOp* ::= +

The LL(1) JASON Grammar (continued)

55 *AddOp ::=* **-**
56 *MultOp ::=* *****
57 *MultOp ::=* /

FIRST set for JASON

FIRST(1) = FIRST(*Header*) = { **program** }

FIRST(2) = { **program** }

FIRST(3) = FIRST(*VarDecls*) = FIRST(*VarDecl*)

FIRST(4) = = FIRST(*VarDecl*) = FIRST(*DataType*)

FIRST(5) = FOLLOW(*VarDecls*) = FIRST(ProcDecls)
 = FIRST(*ProcDecl*) ∪ FOLLOW(*ProcDecls*)

FIRST(6) = FIRST(*DataType*) = { **real**, **integer** }

FIRST(7) = { **real** }

FIRST(8) = { **integer** }

FIRST(9) = { **identifier** }

FIRST set for JASON (continued)

FIRST(10) = { **,** }

FIRST(11) = FOLLOW(*MoreIdList*) = FOLLOW(*IdList*)
 = FOLLOW(*VarDecl*)

FIRST(12) = FIRST(*ProcDecl*) = FIRST(*ProcHeader*) = { **procedure** }

FIRST(13) =FOLLOW(*ProcDecls*)=FIRST(*Block*) = {**begin**}

FIRST(14) = FIRST(*ProcHeader*) = { **procedure** }

FIRST(15) = { **procedure** }

FIRST(16) = { **(** }

FIRST(17) = FOLLOW(*ParamList*) = { ; }

FIRST set for JASON (continued)

FIRST(18) = FIRST(*ParamDecl*) = FIRST(*ParamDecl*)
= FIRST(*DataType)* = { **integer, real** }

FIRST(19) = { **;** }

FIRST(20) = FOLLOW(*MoreParamDecls*) =
FOLLOW(*ParamDecls*) = { **)** }

FIRST(21) = FIRST(*DataType*) = { **integer**, **real** }

FIRST(22) = { **begin** }

FIRST(23) = FIRST(*Statement*)

FIRST(24) = { **;** }

FIRST(25) = FOLLOW(*MoreStatements*) =
FOLLOW(*Statement*)

FIRST set for JASON (continued)

FIRST(26) = { **read** }

FIRST(27) = { **set** }

FIRST(28) = { **write** }

FIRST(29) = { **if** }

FIRST(30) = { **while** }

FIRST(31) = { **until**}

FIRST(32) = { **call** }

FIRST(33) = FOLLOW(*Statement*)

FIRST(34) = { **else** }

FIRST(35) = FOLLOW(*ElseClause*) = { **endif** }

FIRST(36) = { **(** }

FIRST set for JASON (continued)

FIRST(37) = FOLLOW(*ArgList*) = FOLLOW(*Statement*)

FIRST(38) = { **identifier** }

FIRST(39) = { **,** }

FIRST(40) = FOLLOW(*MoreArgs*) = { **)** }

FIRST(41) = FIRST(*Expression*)

FIRST(42) = { **=** }

FIRST(43) = { **!** }

FIRST(44) = { **>** }

FIRST(45) = { **<** }

FIRST(46) = FIRST(*Term*) = FIRST (*Factor*)

FIRST(47) = FIRST(*AddOp*) = { +, **-** }

FIRST set for JASON (continued)

FIRST(48) = FOLLOW(*MoreExpression*)

FIRST(49) = FIRST(*Factor*)

FIRST(50) = FIRST(MultOp) = (*****, / }

FIRST(51) = FOLLOW(*MoreTerm*)

FIRST(52) = {**identifier**)

FIRST(53) = { **constant** }

FIRST(54) = { **+** }

FIRST(55) = { **-** }

FIRST(56) = { ***** }

FIRST(57) = { **/** }

Determining the FIRST set for JASON

Let's determine the FIRST sets for the nonterminals:

FIRST(3) = FIRST(4) = FIRST(*VarDecl*) = FIRST(*DataType*) = { **integer**, **real** }

FIRST(23) = FIRST(Statement)= FIRST(26)∪ FIRST(27) ∪ FIRST(28) ∪ FIRST(29) ∪ FIRST(30) ∪ FIRST(31) ∪ FIRST(32) ∪ FIRST(33) = { **read**, **set**, **write**, **if**, **while**, **until**, **call**} ∪ FOLLOW(Statement)

FIRST(41) = FIRST(46) = FIRST(49) = FIRST(*Expression*) = FIRST(*Term*) = FIRST(*Factor*) = { **identifier**, **constant** }

Determining the FIRST set for JASON (continued)

Now we must find the necessary FOLLOW sets:

FIRST(5) = FIRST(*ProcDecl*) U FOLLOW(*ProcDecls*) = { **procedure** } U FOLLOW (*DeclSec*) = { **procedure**, **begin** }

FIRST(11) = FOLLOW(*IdList*) = FOLLOW(*VarDecl*) = { **integer**, **real**, **procedure**, **begin** }

FIRST(25) = FOLLOW(33) = FOLLOW(37) = FOLLOW(*Statement*) = FOLLOW(*MoreStatements*) = { **;** , **end**, **endif**, **endwhile, enduntil, else** }

FIRST(48) = FOLLOW(*MoreExpression*) = FOLLOW(*Condition*) = { **;**, **end**, **endif**, **endwhile**, **enduntil**, **else**, **then**, **do** }

FIRST(51) = FIRST(*MoreExpression*) U FOLLOW(*Expression*) = {+, **;**, **end**, **endif**, **endwhile**, **enduntil**, **else**, **then**, **do** }

Implementing the Parse Table

```
// The production table for predictive parsing.
// The nonzero entries are the production numbers for a
// particular nonterminal matched with a lookahead token
// Zero entries means that there is no such production
// and it is a parsing error.
const int           prodtable[][numtokens+3] = {
/*Program*/   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/*Header*/    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/*DeclSect*/  { 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 3,
                0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

Implementing the parse table (continued)

```
/*VarDecls*/  { 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 5,
                0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/*VarDecl*/   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0,
                0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/*DataType*/  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0,
                0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/*IdList*/    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 0},
/*MoreIdList*/{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0,11,10, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

The Parsing Algorithm

Processing context-free expressions requires the use of a stack. The Parsing algorithm uses a stack:

```
Place the start symbol in a node and push it onto the stack.
Fetch a token
REPEAT
    Pop a node from the stack
    IF it contains a terminal, match it to the current token (no    match
    indicates a parsing error) and fetch another token
    ELSE IF it contains a nonterminal, look it up in the production table
    using the nontermina and the current token.  Place the variables in
    REVERSE order on the stack
UNTIL the stack is empty
```

Recursive-Descent Parsing

- Recursive-descent parsing is a top-down parsing technique which shows a series of recursive procedures to parse the program
- There is a separate procedure for each individual nonterminal.
- Each procedure is essentially a large if-then-else structure which looks for the appropriate tokens when the grammar requires a particular terminal and calls another procedure recursively when the grammar requires a nonterminal.

Recursive descent parsing of JASON

```
class parser      : scanner   {
public:
  parser(int      argcount, char    *args[]);
  parser(void);
  void      ProcProgram(void);
private:
  void      ProcHeader(void);
  void      ProcDeclSec(void);
  … …
  void      error(char  message[], int linenum);
  tokentype thistoken;
  int             tabindex, level;
};
```

```
symboltable       st;

// main() - Just a simple-minded driver
int   main(int argc, char *argv[])
{
  parser          p(argc, argv);

  p.ProcProgram();
  st.dump();
  return(0);
}

parser::parser(int argcount, char   *args[])
                          : scanner (argcount,args)
{
      level = 0;
      thistoken = gettoken(tabindex);
}
```

```
void  parser::ProcProgram(void)
{
  level++;
  cout << level << '\t' << "Program" << endl;
  //  Program ::= Header      DeclSec     Block .
  ProcHeader();
  ProcDeclSec();
  ProcBlock();
  if (thistoken != tokperiod)
      error("Expected \".\"", linenum);
  getchar();
  ++level;
}
```

```
void  parser::ProcHeader(void)
{
  level++;
  cout << level << '\t' << "Header" << endl;

  // Header ::= program identifier  ;
  if (thistoken != tokprogram)
      error("Expected \"PROGRAM\"", linenum);
  cout << level+1 << '\t' << "program" << endl;
  thistoken = gettoken(tabindex);

  if (thistoken != tokidentifier)
      error("Expected identifier", linenum);
  cout << level+1 << '\t' << "identifier"
                << endl;
  thistoken = gettoken(tabindex);
  if (thistoken != toksemicolon)
      error("Expected \";\"", linenum);
  cout << level+1 << '\t' << ";" << endl;
  thistoken = gettoken(tabindex);

  --level;
}
void  parser::ProcDeclSec(void)
{
  level++;
  cout << level << '\t' << "DeclSec" << endl;

  // DeclSec ::= VarDecls ProcDecls
  ProcVarDecls();
  ProcProcDecls();

  --level;
}
```