## Chapter 4 Top-Down Parsing

- Study Goals:
- Master
  Recognition of LL(1) grammar, Construction of Recursive-descent parsing, LL(1) parsing
- Understand
  First set, Follow set, LL(1) grammar
- Know
  Backtracking Parsing, Error Recovery in Top-Down parsers, Syntax Tree Construction in Top-down parsers

## Top-Down Parsing

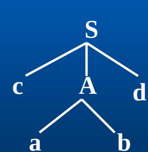
- Definition
  Parsing begins with the start symbol of grammar and tries to find out the derivation of the input string tracing out the steps in a leftmost derivation

  The construction of Parse Tree:
  Start symbol of grammar is the root, parse tree is constructed from the root to leaves in preorder, the leaves of the parse tree are just input string of tokens

Example G： S → cAd
A → ab
A → a

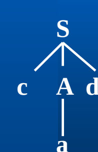
Top-down parsing of string “cabd”

Derivation: S =>cAd =>cabd




## The Key of Top-down Parsing

Another top-down parsing process for input string “cabd” is:



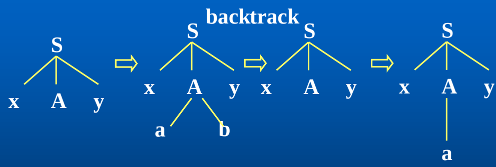

Fail, because a derivation step has selected a wrong production A→a

- So the problem of top-down parsing is the choice of a production for a derivation step
- If the leftmost nonterminal to be replaced is B, there are totally n productions of B: B→A1|A2|…|An, how can we determine which one to use?

## Categories of Top-down parsing

1. Backtracking Parsing
- A nonterminal has more than one productions
- and basing on the current input symbol, the parser can’t determine which one to choose
- it must try different possibilities, backing up an arbitrary amount in the input if one possibility fails.

Example: Grammar :S→xAy  A→ab|a
Top-down parsing of string “xay”




2. Predictive parsing
Parser attempts to predict the next construction in the input string using one or more lookahead tokens

- Two Kinds of Predictive Parsing
  - Recursive-descent parsing
  - LL(1) parsing



## 4.1 Predictive Parsing

1 The Condition of Predictive Parsing
2 The Definition of Lookahead Sets
3 The Definition of LL(1) Grammar

## 1 The Condition of Predictive Parsing

- Parsing of the input string begins with the start symbol of the grammar
- if it can uniquely determine which production to use next in derivation basing on the current token of the input, parsing is predictive

- The Condition of Predictive Parsing
  Predictive parsing require that the grammar must be a LL(1) grammar
- What is LL(1) grammar?
  It’s definition depends on the definition of lookahead sets--First set and Follow set

## 2 The Definition of Lookahead Sets

1) **First Sets**

- Definition

  G=($V_N$, $V_T$, P, S) is a grammar, $\beta \in (V_N \cup V_T)^*$

  FIRST(β) = { a $\in V_T$ | β ⇒* a......}

  if β⇒* ε then ε ∈FIRST(β)

  Intuitively, the first set of string β is the set of first terminals(including ε) that can be derived from β

- Example G[S]:

| | |
|---|---|
| S→Ap | FIRST(Ap)={a,c} |
| S→Bq | FIRST(Bq)={b,d} |
| A→a | FIRST(a)={a } |
| A→cA | FIRST(cA)={c} |
| B→b | FIRST(b)={b} |
| B→dB | FIRST(dB)={d} |

If there are more than one productions of nonterminal A: A-> α | β|…,but FIRST(α) ∩ FIRST(β) ∩ …= Φ, this grammar can be predictive parsed. Basing on the current input symbol which production to choose is determined

2) **Follow Sets**

- Definition

  G=($V_T$, $V_N$, S,P) is a grammar, $A \in V_N$ ,

  FOLLOW(A)={a $\in V_T$ |S=>*…Aa…},

  if S=>* …A, then $ ∈FOLLOW(A)

  ($ is used to mark the end of the input)

  Intuitively, the follow set of nonterminal A is the set of terminals(include $) following A in all sentential form of the grammar

- Example $G_3$[S]:

  S→aA|d

  A→bAS|ε

  - S=>*aA ,$ ∈FOLLOW(A)

    S=>*abAS=>*abAaA, a ∈FOLLOW(A)

    … =>*abAd , d ∈FOLLOW(A)

    FOLLOW(A)={$,a,d}

  - S=>*S,$ ∈FOLLOW(S)

    S=>aA=>abAS=>abbASS=>abbASaA

    …=>abbASd

    FOLLOW(S)={$,a,d}

There are two productions of nonterminal A:

A→bAS and A→ε, let the current input symbol is “x”

- If x ∈FIRST(bAS)={b} ,then choose A→bAS for derivation
- If x ∈FOLLOW(A)={$,a,d } , then choose A→εfor derivation
- Because FIRST(bAS)∩FOLLOW(A)=ф, which production to choose is determined

S→AB|bC

A→a|ε

B→aD|ε

C→AD|b

D→aS|c

aac

## 3 The Definition of LL(1) Grammar

- LL(1) Grammar

  A grammar is LL(1) if the following conditions are satisfied:

1. For each production $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$
   $First(\alpha_i) \cap First(\alpha_j) = \Phi$ for all i and j, $1 \le i,j \le n, i \ne j$
2. For each nonterminal A such that First(A) contains ε, $First(A) \cap Follow(A) = \Phi$

Example G[S]：

| | |
|---|---|
| S→aAS | First(aAS) = {a} |
| S→b | First(b) = {b} |
| A→bA | First(bA) = {b} |
| A→ε | First(ε) ={ε} |
| | First(A)={b, ε} Follow(A)= {a,b} |

Because First(A)∩Follow(A)={b}≠Φ,

G[S] is not a LL(1) grammar, when the leftmost nonterminal to be replaced is A and the current input symbol is "b", parsing can't determine which productions of A to choose :A→bA or A→ε

## 4.2 Recognition of LL(1) Grammar

Recognition of LL(1) grammar has the following four steps:

1. Compute the set of nullable nontermianls
2. Compute FIRST(α) for the right-hand side string α of each production
3. Compute FOLLOW(A) for each nonterminal A
4. Recognize basing on the definition of LL(1)

## 1 Compute the set of nullable nontermianls

- Nullable

  A nonterminal A is nullable if there exists a derivation $A \Rightarrow^* \varepsilon$
- Algorithm

  Let S is the set of nullable nonterminals

1. First, $S=\{ A_j \mid A_j \to \varepsilon$ is a production$\}$
2. For each production p: $A_p \to X_1 \ldots X_n$, if $X_1 \ldots X_n \in S$, then $S := S \cup \{A_p\}$
3. Repeat step 2 until there is no change to S

Example G[S]:

S→AB|bC
A→b|ε
B→aD|ε
C→AD|b
D→aS|c

| | Set S |
|---|---|
| Origin | {A,B} |
| Pass 1 | {A,B,S} |
| Pass 2 | {A,B,S} |

The set of nullable nonterminals is {A,B,S}

## 2 Compute FIRST(α) for the right-hand string α of each production

- Algorithm for computing First(A) for each grammar symbol A ($A \in V_T \cup V_N$)
- Algorithm for computing First(α) for a string α

## Algorithm for computing First(A) for each grammar symbol A ($A \in V_T \cup V_N$)

1. For all $a \in V_T$ do First(a)={ a }
2. For all $A \in V_N$ ,if $A \Rightarrow^* \varepsilon$ then First(A)={ε} else First(A)={ }
3. For each production A→X1…Xj…Xn ,

First(A)=First(A)∪SectionFirst(X1…Xj…Xn)

4. Repeat step 3 until there is no change to any First set

---

SectionFirst(X1…Xj…Xn)

$= (First(X_1) - \{\varepsilon\}) \cup (First(X_2) - \{\varepsilon\}) \cup \ldots \cup (First(X_j) - \{\varepsilon\}) \cup First(X_{j+1})$

$X_{j+1}$ is the first symbol that is not nullable in the right-hand of production

- If $X_1$ is not nullable, then SectionFirst(X1…Xj…Xn) =First($X_1$)
- If X1…Xn are all nullable, then SectionFirst(X1…Xn)

$= (First(X_1) - \{\varepsilon\}) \cup (First(X_2) - \{\varepsilon\}) \cup \ldots \cup (First(X_n) - \{\varepsilon\}) \cup \{\varepsilon\}$

---

G[S]: S→AB|bC
A→b|ε
B→aD|ε
C→AD|b
D→aS|c

The set of nullable nonterminals is {A,B,S}

| | Origin | Pass 1 | Pass 2 | Pass 3 |
|---|---|---|---|---|
| S | ε | ε b | ε b a | ε b a |
| A | ε | ε b | ε b | ε b |
| B | ε | ε a | ε a | ε a |
| C | | b | b a c | b a c |
| D | | a c | a c | a c |
| a | a | a | a | a |
| b | b | b | b | b |

---

## Algorithm for computing First(α) for a string $\alpha = X_1X_2 \ldots X_n$

1. If $X_1$ is not nullable, then FIRST(α)=FIRST($X_1$)
2. if Xj ($1 \le j < n$) is nullable

then FIRST(α)

$=(FIRST(X_1) - \{\varepsilon\}) \cup \ldots \cup (FIRST(X_j) - \{\varepsilon\})$

$\cup FIRST(X_{j+1})$

3. if Xi ($1 \le i \le n$) are all nullable, then

$FIRST(\alpha) = (FIRST(X_1) - \{\varepsilon\}) \cup \ldots \cup (FIRST(X_n) - \{\varepsilon\}) \cup \{\varepsilon\}$

---

Example

G[S] S→AB|bC
A→b|ε
B→aD|ε
C→AD|b
D→aS|c

First set for nonterminals

First(S)={a,b, ε} First(A)={b, ε}

First(B)={a, ε} First(C)={a,b,c}

First(D)={a,c}

First sets for strings in the right-hand side of productions

S→AB FIRST(AB)= (FIRST(A) - {ε}) ∪ (FIRST(B) - {ε}) ∪{ε}={a,b,ε}

S→bC FIRST(bC)= {b}

A→ε FIRST(ε)= {ε}

A→b FIRST(b)= {b}

C→AD FIRST(AD)=(FIRST(A)-{ε})∪FIRST(D) ={b,a,c}

D→aS FIRST(aS)= {a}

---

## 3 Compute FOLLOW(A) f… al A

because S=>*S, \$∈FOLLOW(S)

1.S is the start symbol, Follow(S)={\$};for all $A \in V_N$, and A≠S,Follow(A)={ };

2.For each production B→αAγ, for each A that is a nonterminal do

Follow(A)=Follow(A)∪(First(γ) -{ε})

if ε ∈ First(γ) then add Follow(B) to Follow(A)

if b∈FOLLOW(B), then S=>*…Bb…,
because B→αAγ, and γ =>*,
so S=>*…Bb…=>…αAγ b=>…αAb…,
that is S=>*…αAb…, b∈FOLLOW(A)

G[S]:
[1]S->AB
[2]S->bC
[3]A->b
[4]A->ε
[5]B->aD
[6]B->ε
[7]C->AD
[8]C->b
[9]D->aS
[10]D->c

First sets of nonterminals are:

First(S)={a,b, ε} First(A)={b, ε}

First(B)={a, ε} First(C)={a,b,c}

First(D)={a,c}

| | Origin | Pass 1 | Pass 2 |
|---|---|---|---|
| S | $ | $ | $ |
| A | | a $ c | a $ c |
| B | | $ | $ |
| C | | $ | $ |
| D | | $ | $ |

- Compare First set with Follow set
  - ε is an element of First set but never of Follow set
  - First set is defined for nonterminals and strings of terminals and nonterminals , while Follow set is defined only for nonterminals
  - The definition of Follow set works "on the right" of production, while the definition of the First set works "on the left"

## 4 Recognize basing on the definition of LL(1)

Condition of LL(1) :

1. For each production $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$, $First(\alpha_i) \cap First(\alpha_j) = \Phi$, for all i and j, $1 \le i, j \le n$, $i \ne j$
2. For every nonterminal A such that First(A) contains ε, $First(A) \cap Follow(A) = \Phi$

G[S]:
S→AB|bC
A→b|ε
B→aD|ε
C→AD|b
D→aS|c

| | nullable | First | Follow |
|---|---|---|---|
| S | yes | {a,b, ε} | {$} |
| A | yes | {b, ε} | {a,c,$} |
| B | yes | {a, ε} | {$} |
| C | no | {a,b,c} | {$} |
| D | no | {a,c} | {$} |

- S->AB |bC : First(AB)={a,b, ε}, First(bC)={b}

First(AB) ∩First(bC) ={b}≠ф

- C->AD | b : First(AD)={b,a,c}, First(b)={b}

First(AD)∩First(b) ={b}≠ф

- This grammar is not LL(1)

## 4.3 Non LL(1) grammar to LL(1) grammar

- Non LL(1) grammar
  - If a grammar has either left factor or left recursion, or both, then it must be non LL(1) grammar
  - Whereas, a grammar that doesn't have left factor and left recursion is not always a LL(1) grammar

1. Left Factor

- Left factor is two or more grammar rule choices sharing a common prefix string, as in the rule
  $A \to \alpha\beta|\alpha r$
- Because First(αβ)∩First(α r)≠ф, so it's not LL(1)

2. Left Recursion

A grammar is left recursive if its productions have the following forms:

a) $A \to A\beta$

b) $A \to B\beta \quad B \to A\alpha$

- a) is called immediate left recursion, where the left recursion occurs only within the production of a single nonterminal
- b) is called indirect left recursion, where $A \Rightarrow B\beta \Rightarrow A\alpha\beta$, that is $A \Rightarrow^{+} A\ldots$

Take immediate left recursion for example,

if there are productions: $A \to A\alpha \mid A \to \beta$

where $\alpha$ and $\beta$ are arbitrary strings

because First( $A\alpha$ )$\supseteq$First( $\beta$ )

so it is not a LL(1) grammar

- Techniques for rewriting non LL(1) grammar into LL(1)
  - Left recursion removal
  - Left factoring

Notice:There is no guarantee that the application of these techniques will turn a grammar into LL(1) grammar

## 1 Left Recursion Removal

Immediate left recursion removal

- Simple case
  $A \to A\alpha \mid \beta$ rewrite this rule into
  $A \to \beta A', A' \to \alpha A' \mid \varepsilon$
- General case
  $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
  rewrite this rule into：
  $A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
  $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

## 2 Left Factoring

- Simple case
  Rewrite the rule $A \to \alpha\beta \mid \alpha r$ as
  $A \to \alpha(\beta \mid r)$，let $A'$ represents $\beta \mid r$, we get
  $A \to \alpha A' \qquad A' \to \beta \mid r$
  $\alpha$ must be the longest string shared by the right-hand sides.
- General case
  $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$,
  rewrite the rule as:
  $A \to \alpha A' \qquad A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

- Example

G1：$S \to aSb \mid aS \mid \varepsilon$

Left factoring:

$S \to aS(b \mid \varepsilon) \mid \varepsilon$

The same as :

$S \to aSS' \mid \varepsilon$

$S' \to b \mid \varepsilon$

## 4.4 Top-Down Parsing by Recursive-Descent

- Main Idea of Recursive-Descent
- Define a procedure for each nonterminal A that will recognize A
- The right-hand side of the grammar rule for A specifies the structure of the code for this procedure
  - Terminals correspond to matches of the input
  - Nonterminals correspond to calls to other procedures
  - Choices correspond to alternatives(case or if statement) within the code

We will talk about:

- General method for constructing Recursive-Descent parser
- A simpler method based on EBNF

## General method for constructing Recursive-Descent parser

1. Determine whether the grammar is LL(1)

Example Grammar G

E→E+T│T
T→T*F│F
F→(E)│i

1) After left recursion removal we get grammar G':

E→TE'    E'→+TE'│ε
T→FT'    T'→*FT'│ε
F→(E)│i

2) First and Follow Set

| | nullable | First | Follow |
|---|---|---|---|
| E | no | { (, i } | { ), $ } |
| E' | yes | { +, ε} | { ), $ } |
| T | no | { (, i } | { +, ), $ } |
| T' | yes | { *, ε} | { +, ), $ } |
| F | no | { (, i } | { *, +,),$} |

E→TE'
E'→+TE'│ε
T→FT'
T'→*FT'│ε
F→(E)│i

G' is LL(1) grammar

## 2 Construct Recursive-Descent Parser for G'

- When a grammar is LL(1), we can construct recursive-descent parser for it .
- The parser consists of a main procedure and a group of recursive procedures, each corresponds to a nonterminal of the grammar

## Construction of procedure for a nonterminal

- Sub procedures used :
  - match is a procedure that matches the current next token with its parameter, advances the input if it succeeds, and declares error if it does not
  - error is a procedure that prints an error message and exit
- Variable used:
  - TOKEN is a variable that keeps the current next token in the input

1) If productions of nonterminal U are
U->x1 | x2 |…|xn, and x1,...xn≠ ε, then the code for procedure U is as follow:
if TOKEN in First(x1) then p_x1
else if TOKEN in First(x2) then p_x2
else …
….
if TOKEN in First(xn) then p_xn
else ERROR

2) If a production of U is U-> ε, then rewrite code
if TOKEN in First(xn) then p_xn else ERROR
into
if TOKEN in First(xn) then p_xn
else if TOKEN not in Follow(U) then ERROR

3) The code for p_x where x=y1y2…yn is:
begin p_y1;p_y2;…;p_yn end
if yi∈$V_N$ then p_yi is the call of procedure yi;
otherwise, if yi∈$V_T$ then p_yi is match(yi)

Example:Recursive-descent parser for G'

E→TE '　　E'→+TE'│ε
T→FT '　　T'→*FT'│ε
F→(E)│i

```
(1) program MAIN;                    /* main */
     begin
       GETNEXT (TOKEN);
       E ;                           /* call E */
       if TOKEN ≠'$' then ERROR
    end.
```

```
(2) procedure E;                     /*E→TE'*/
     begin
       T;                            /*call T*/
       E'                            /*call E'*/
    end;
(3) procedure T;                     /*T→FT' */
     begin
       F;                            /*call F*/
       T'                            /*call T'*/
     end;
```

```
(4) procedure E';                    /*E'→+TE'│ε*/
     begin
       if TOKEN='+' then             /*E'→+TE'*/
         begin
           match('+');
           T;                        /*call T*/
           E'                        /*call E'*/
         end
       else                          /*E'->ε*/
       if TOKEN≠')' and TOKEN≠'$' then ERROR
    end;
```

```
(5) procedure T';                    /* T'→*FT'│ε */
     begin
       if TOKEN = '*' then           /* T'→*FT' */
         begin
           match('*');
           F;                        /* call F */
           T'                        /* call T' */
         end
       else                          /* T'→ε*/
    if TOKEN≠'+' and TOKEN≠')' and TOKEN≠'$'
    then ERROR
end;
```

```
(6)  procedure  F;                    /* F→(E)│i */
      begin
        if  TOKEN = '('  then              /*F→(E) */
          begin
            match('(');
            E;                             /* call E */
            match(')')
          end
        else                               /* F→i  */
            if  TOKEN='i'  then match('i')
            else error;
     end;
```

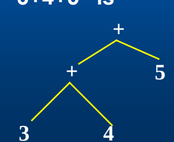
Note:

- Left factoring and left recursion removal change the grammar, these changes cause the complication of the parser
- Left factoring and left recursion removal can obscure the semantics of the language structure (for example, they obscure the associativity in arithmetic expressions)

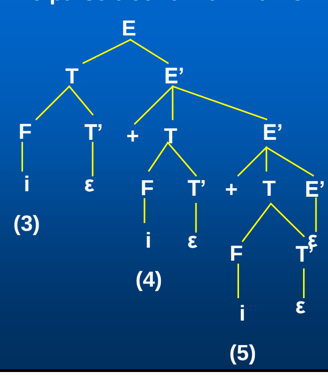
Example

E→TE'
E'→+TE'│ε
T→FT'
T'→*FT'│ε
F→(E)│i

The syntax tree for "3+4+5" is




The parse tree for "3+4+5" is




- A simpler method based on EBNF

- Notations of EBNF (extended BNF)
- { } is used to express repetition

  Example:

  A->Aα|β is written as

  A->β{α}

- [ ] is used to surround options

  Example:

  if-stmt->if (exp) stmt

  | if (exp) stmt else stmt

  is written as

  if-stmt->if (exp) stmt [else stmt]

Example G

E→E+T│T
T→T*F│F
F→(E)│i

rewrite G into EBNF as G':

E-> T {+ T}

T-> F {* F}

F->(E) | i

## Construct Recursive-Descent Paring for G'

- [ ] is translated into a test in the code
- { } is translated into the code for a while loop

if-stmt->if (exp) stmt [else stmt]

can be translated into the procedure:

```
Procedure ifStmt;
begin
     match('if');
     match('(');
     Exp;
     match(')');
     Stmt;
     if token='else' then
             match('else');
             Stmt;
     end if;
end ifStmt;
```

E-> T {+ T}

```
(1)  procedure  E;
      begin
        T;
        while token='+' do
                match('+');
                T;
        end while;
        end
```

T-> F {* F}

```
(2)  procedure  T;
      begin
        F;
        while token='*' do
                match('*');
                F;
        end while;
        end
```

## Action for Constructing a Syntax Tree

Sub functions used

- MakeOpNode that receives an operator token as a parameter and returns a newly constructed syntax tree node
- leftChild(t):=p or rightChild(t):=p indicates the assignment of a syntax tree p as a left or right child of a syntax tree t

```
function  E:syntaxTree;            /* E-> T {+ T} */
var temp,newtemp:syntaxTree;
begin
        temp:=T;
        while token='+' do
                newtemp:=makeOpNode(token);
                match('+');
                leftChild(newtemp):=temp;
                rightChild(newtemp):=T;
                temp:=newtemp;
        end while;
        return temp;
end
```

```
function  ifStmt: syntaxTree;
                        /* if-stmt->if (exp) stmt [else stmt] */
var temp :syntaxTree;
begin
    match('if');
    match('(');
    temp:=makeStmtNode('if');
    testChild(temp):=Exp;
    match(')');
    thenChild(temp):=Stmt;
    if token='else' then
        match('else');
        elseChild(temp):=Stmt;
    else
        elseChild(temp):=nil;
    end if;
end
```

Note:

- The method of turning grammar rules in EBNF into code in quite powerful
- But it may be difficult to convert a grammar originally written in BNF into EBNF form

- Character of recursive-descent parsing

- Merit:
  - Versatile, powerful
  - Flexible by allowing the programmer to adjust the scheduling of the actions
  - Suitable for hand-generated parsers
- Pitfall:
  - Care must be taken in scheduling the actions within the code
  - Recursive calls result in slowness and much more space

Quiz

What are the first and follow sets of S?

S → A ( S ) B | ϵ
A → S | S B | x | ϵ
B → S B | y

- A First: {x, '('}, Follow: {$, y, x}
- B First: {x, y, '(', ϵ}, Follow: {$, y, x, '(', ')'}
- C First: {x, y, '(', ϵ}, Follow: {y, x, '(', ')'}
- D First: {x, y, '(', ϵ}, Follow: {$, '(', y}
- E First: {x, ϵ}, Follow: {$, y, x, '(', ')'}
- F First: {x, y, '('}, Follow: {$, y, x, '(', ')'}

## Questions: Predictive Parsing

- Left-factor the grammar
  S→aSb|aS|ε

S→aS(b|ε)|ε

S→aSS'|ε
S'→b|ε

In-video Quiz: Left Recursion

Choose the grammar that correctly eliminates left recursion from the given grammar: E → E + T | T
T → id | (E)

A
E → E + id | E + (E)
| id | (E)

B
E → TE'
E' → + TE' | ε
T → id | (E)

C
E → E' + T | T
E' → id | (E)
T → id | (E)

D
E → id + E | E + T | T
T → id | (E)

## 4.5 LL(1) Parsing

- The meaning of LL(1)
  - The first "L" refers to the fact that it processes the input from left to right
  - The second "L" refers to the fact it traces out a leftmost derivation for the input string
  - The number "1" means that it uses only one symbol of input to predict the direction of the parse

## 4.5.1 The Basic Method of LL(1) Parsing

- LL(1) parsing uses an explicit stack rather than recursive calls to perform a parse
  - Stack stores the symbols waiting to be matched in parsing
  - Parsing begins by pushing the start symbol into the stack
  - The stack and the input will be empty at the end of a successful parsing

– A schematic for a successful LL(1) parsing is:

| | | |
|---|---|---|
| $StartSymbol | InputString$ | |
| … | … | |
| $ | $ | accept |

'$' is used to mark the bottom of the stack and the end of the input string

- Example
  S->(S)S| ε

The actions of LL(1) parser for string "( )"

| | Parsing stack | Input | Action of parser |
|---|---|---|---|
| 1 | $ S | ( )$ | S->(S)S |
| 2 | $ S ) S ( | ( )$ | match |
| 3 | $ S ) S | )$ | S-> ε |
| 4 | $ S ) | )$ | match |
| 5 | $ S | $ | S-> ε |
| 6 | $ | $ | accept |

## 4.5.2 The LL(1) Parsing Table

- Function
  Expressing production choices for a nonterminal to use at the appropriate parsing step
- Example

| M[N,T] | ( | ) | $ |
|---|---|---|---|
| S | S->(S)S | S->ε | S->ε |

- LL(1) Parsing table
  - A two-dimensional array indexed by nonterminals and terminals
  - M[N,t] is a production choice of N to use at the appropriate parsing step, based on the current input token t（terminal or '$')
  - If M[N,t] remain empty after construction, it represent potential errors that may occur during a parse

- The Construction of Parsing Tables

Repeat the following two steps for each nonterminal A and production choice A→ α

1. For each token 'a' in First(α), add A→ α to the entry M[A,a]
2. if εis in First(α), for each element 'a' of Follow(A) (token or $), add A→ α to M[A,a]

Example :Grammar of arithmetic expression

E→E+T│T

T→T*F│F

F→(E)│i

（1）After left recursive removal gets G'

E→TE '

E'→+TE'│ε

T→FT '

T'→*FT'│ε

F→(E)│i

E→TE '

E'→+TE'│ε

T→FT '

T'→*FT'│ε

F→(E)│i

（2）First and Follow set for nonterminals

| | nullable | First | Follow |
|---|---|---|---|
| E | no | {(,i} | {),$} |
| E' | yes | {+, ε} | {),$} |
| T | no | {(,i} | {+,),$} |
| T' | yes | {*, ε} | {+,),$} |
| F | no | {(,i} | {*,+,),$} |

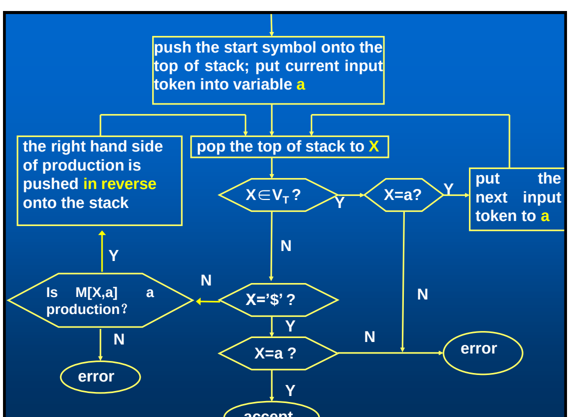
（3）Construct parsing table

| | i | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E→TE' | | | E→TE' | | |
| E' | | E'→+TE' | | | E'→ ε | E'→ ε |
| T | T→ FT' | | | T→ FT' | | |
| T' | | T'→ ε | T'→* FT' | | T'→ ε | T'→ ε |
| F | F→ i | | | F→ (E) | | |

- LL(1) grammar

  A grammar is a LL(1) grammar if the associated LL(1) parsing table has at most one production in each table entry.

## 4.5.3 The LL(1) parsing algorithm




Parsing process of string "i+i*i$"

| | i | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E→TE' | | | E→TE' | | |
| E' | | E'→+TE' | | | E'→ ε | E'→ ε |
| T | T→ FT' | | | T→ FT' | | |
| T' | | T'→ ε | T'→* FT' | | T'→ ε | T'→ ε |
| F | F→ i | | | F→ (E) | | |

| Step | Stack | Input | Action |
|---|---|---|---|
| 1 | $E | i+i*i$ | E→TE' |
| 2 | $E'T | i+i*i$ | T→FT' |
| 3 | $E'T'F | i+i*i$ | F→i |
| 4 | $E'T'i | i+i*i$ | match i |
| 5 | $E'T' | +i*i$ | T'→ ε |
| 6 | $E' | +i*i$ | E'→+TE' |
| 7 | $E'T+ | +i*i$ | match + |

| | i | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E→TE' | | | E→TE' | | |
| E' | | E'→+TE' | | | E'→ ε | E'→ ε |
| T | T→ FT' | | | T→ FT' | | |
| T' | | T'→ ε | T'→* FT' | | T'→ ε | T'→ ε |
| F | F→ i | | | F→ (E) | | |

| | | | |
|---|---|---|---|
| 8 | $E'T | i*i$ | T→FT' |
| 9 | $E'T'F | i*i$ | F→i |
| 10 | $E'T'i | i*i$ | match i |
| 11 | $E'T' | *i$ | T'→ *FT' |
| 12 | $E'T'F* | *i$ | match * |
| 13 | $E'T'F | i$ | F→i |
| 14 | $E'T'i | i$ | match i |
| 15 | $E'T' | $ | T'→ ε |
| 16 | $E' | $ | E'→ ε |
| 17 | $ | $ | accept |

## 4.6 Error Recovery in Top-down Parsers

- A parser must determine whether a program is syntactically correct or not
- The response of a parser to syntax errors is often a critical factor in the usefulness of a compiler

- Normal syntactical errors
- Start and following symbols error

  The start and following symbol of program, expressions and statements

for example, the TINY language

| | start symbol | following symbol |
|---|---|---|
| statement | if, repeat, identifier, read, write | ; else end until |
| exp | (, number, identifier | ) then ; |

- Identifiers and constants error

  Such as "const", "var", "procedure" is not followed by identifiers
- Parenthesis matching error

  Such as, begin—end, if—then don't match
- Symbols error

  Such as symbol in assignment is not ':='

- Error handing
- Give a meaningful error message
- Pick a likely place to resume the parse. A parser should always try to parse as much of the code as possible, in order to find as many real errors as possible during a single translation

- Categories of Error Handing
  - Error Recovery: after an error has occurred, the parser picks a likely place to resume the parsing
  - Error Repair: the parser attempts to infer a correct program from the incorrect one given

## 4.6.1 Error Recovery in Recursive-Descent Parsers

- Panic Mode

  In complex situations, the error handler will consume a possibly large number of tokens in an attempt to find a place to resume parsing

- The Basic Mechanism of Panic Mode
  - Provide each recursive procedure with an extra parameter consisting of a set of synchronizing tokens
  - As parsing proceeds, tokens that may function as synchronizing tokens are added to this set as each call occurs
  - If an error is encountered, the parser scans ahead, throwing away tokens until one of the synchronizing set of tokens is seen in the input

Follow sets are important candidates for synchronizing tokens

Example

Assume SynchSetS is the synchronizing set of S

| | |
|---|---|
| S->id:=$E_1$ | |
| S->if $E_2$ then $S_1$ else $S_2$ | SynchSet$E_1$=SynchSetS |
| S->while $E_3$ do $S_3$ | SynchSet$E_2$={then} |
| S->repeat $S_4$ until $E_4$ | SynchSet$S_1$={else} |
| S->begin SL end | SynchSet$S_2$=SynchSetS |
| | SynchSet$E_3$={do} |
| | SynchSet$S_3$=SynchSetS |
| | SynchSet$S_4$={until} |
| | SynchSet$E_4$=SynchSetS |
| | SynchSetSL={end} |

- First sets may also be used to prevent the error handler from skipping important tokens that begin major new constructs
- First sets are also important, in that they allow a recursive descent parser to detect errors early in the parse

## Realization of Panic Mode

1) At the beginning of each procedure, check whether the current input token is in FirstSet. If not, scan ahead until find the first token in FirstSet ∪SynchSet, parsing is resumed from this token
2) Before returning from a procedure, check whether the next token is in SynchSet. If not, scan ahead until find the first token in FirstSet ∪SynchSet, parsing is resumed from this token

Example

E-> T {+ T}

T-> F {* F}

F->(E) | i

```
procedure checkinput( firstset, followset );
begin
   if  not (token in firstset) then
          error;
          getToken;
          while  not  (token  in  (firstset  ∪  followset))
          do      getToken;
   end if;
end;
```

```
E-> T {+ T}
procedure  E(synchset);
begin
   checkinput( { (, i }, synchset );
   if not (token in synchset) then
        T(synchset);
        while token=‘+’ do
                match(‘+’);
                T(synchset);
        end while;
        checkinput( synchset, { (, i } );
   end if;
end
```