

# COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Loudon

# 4. Top-Down Parsing

## PART TWO

# Contents

## PART ONE

4.1 Top-Down Parsing by Recursive-Descent

4.2 LL(1) Parsing [\[More\]](#)

## PART TWO

4.3 First and Follow Sets [\[More\]](#)

4.4 A Recursive-Descent Parser for the TINY  
Language [\[More\]](#)

4.5 Error Recovery in Top-Down Parsers

## 4.1 Top-Down Parsing by Recursive-Descent

## **4.2 LL(1) Parsing**

## **4.2.1 The Basic Method of LL(1) Parsing**

# Main idea

- LL(1) Parsing uses an explicit stack rather than recursive calls to perform a parse
- An example:
  - a simple grammar for the strings of balanced parentheses:  
$$S \rightarrow (S) S \mid \varepsilon$$
- The following table shows the actions of a top-down parser given this grammar and the string ( )

# Table of Actions

Steps	Parsing Stack	Input	Action
1	\$S	( ) \$	$S \rightarrow (S) S$
2	\$S)S(	( ) \$	match
3	\$S)S	)\$	$S \rightarrow \epsilon$
4	\$S)	)\$	match
5	\$S	\$	$S \rightarrow \epsilon$
6	\$	\$	accept



# General Schematic

- A top-down parser begins by pushing the start symbol onto the stack
- It accepts an input string if, after a series of actions, the stack and the input become empty
- A general schematic for a successful top-down parse:

\$ StartSymbol	Inputstring\$	
...	...	//one of the
two actions		
...	...	//one of the two actions
\$	\$ accept	

# Two Actions

- **The two actions**
  - **Generate**: Replace a non-terminal  $A$  at the top of the stack by a string  $\alpha$  (in reverse) using a grammar rule  $A \rightarrow \alpha$ , and
  - **Match**: Match a token on top of the stack with the next input token.
- The list of generating actions in the above table:
$$\begin{aligned} S &\Rightarrow (S)S && [S \rightarrow (S) S] \\ &\Rightarrow ( )S && [S \rightarrow \epsilon] \\ &\Rightarrow ( ) && [S \rightarrow \epsilon] \end{aligned}$$
- Which corresponds precisely to the steps in a leftmost derivation of string  $( )$ .
- This is the characteristic of top-down parsing.

## **4.2.2 The LL(1) Parsing Table and Algorithm**

# Purpose and Example of LL(1) Table

- Purpose of the LL(1) Parsing Table:
  - To express **the possible rule choices** for a non-terminal **A** when the **A is at the top of parsing stack** based on the **current input token** (the look-ahead).
- The LL(1) Parsing table for the following simple grammar:

$$S \rightarrow (S) S \mid \epsilon$$

M[N,T]	(	)	\$
S	$S \rightarrow (S) S$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

# The General Definition of Table

- The table is a **two-dimensional array** indexed by non-terminals and terminals
- Containing production **choices to use at the appropriate parsing step** called  $M[N,T]$ 
  - $N$  is the set of non-terminals of the grammar
  - $T$  is the set of terminals or tokens (including \$)
- Any entrances remaining **empty**
  - Representing **potential errors**

# Table-Constructing Rule

- The table-constructing rule
  - If  $A \rightarrow \alpha$  is a production choice, and **there is a derivation  $\alpha \Rightarrow^* a\beta$** , where **a** is a token, then add  $A \rightarrow \alpha$  to the table entry  $M[A, a]$ ;
  - If  $A \rightarrow \alpha$  is a production choice, and **there are derivations  $\alpha \Rightarrow^* \epsilon$  and  $S\$ \Rightarrow^* \beta A a \gamma$** , where  $S$  is the start symbol and **a** is a token (or **\$**), then add  $A \rightarrow \alpha$  to the table entry  $M[A, a]$ ;

# A Table-Constructing Case

- The constructing-process of the following table
  - For the production :  $S \rightarrow (S)S$ ,  $a=(S)S$ , where  $a=($ , this choice will be added to the entry  $M[S, (]$ .
  - Since:  $S \Rightarrow^* (S)S\$$ , rule 2 applied with  $A = S$ ,  $a = )$ , so add the choice  $S \rightarrow \epsilon$  to  $M[S, )]$
  - Since  $S\$ \Rightarrow^* S\$$ ,  $S \rightarrow \epsilon$  is also added to  $M[S, \$]$ .

$M[N,T]$	(	)	\$
<b>S</b>	$S \rightarrow (S) S$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

# Properties of LL(1) Grammar

- **Definition of LL(1) Grammar**
  - A grammar is an LL(1) grammar if the associated LL(1) parsing table has **at most one production in each table entry**
- An LL(1) grammar **cannot be ambiguous**



# A Parsing Algorithm Using the LL(1) Parsing Table

(\* assumes \$ marks the bottom of the stack and the end of the input \*)

push the start symbol onto the top the parsing stack;

while the top of the parsing stack  $\neq$  \$ and  
the next input token  $\neq$  \$ do

if *the top of the parsing stack is terminal a and the next input token = a*

then (\* match \*)

pop the parsing stack;

advance the input;

# A Parsing Algorithm Using the LL(1) Parsing Table

else if *the top of the parsing stack is non-terminal A*  
and *the next input token is terminal a*  
and *parsing table entry  $M[A,a]$  contains production  $A \rightarrow$*

*$X_1X_2...X_n$*

then (\* generate \*)

pop the parsing stack;

for  $i:=n$  downto 1 do

push  $X_i$  onto the parsing stack;

else error;

if *the top of the parsing stack =  $\$$*

and the next input token =  $\$$

then **accept**

else error.

# Example: If-Statements

- The LL(1) parsing table for simplified grammar of if-statements:

**Statement**  $\rightarrow$  if-stmt | other

**If-stmt**  $\rightarrow$  if (exp) statement else-part

**Else-part**  $\rightarrow$  else statement |  $\epsilon$

**Exp**  $\rightarrow$  0 | 1

<b>M[N,T]</b>	<b>If</b>	<b>Other</b>	<b>Else</b>	<b>0</b>	<b>1</b>	<b>\$</b>
<b>Statement</b>	<b>Statement → if- stmt</b>	<b>Statement → other</b>				
<b>If-stmt</b>	<b>If-stmt → if (exp) stateme nt else- part</b>					
<b>Else-part</b>			<b>Else-part → else state ment</b>  <b>Else-part → ε</b>			<b>Else- part → ε</b>
<b>Exp</b>				<b>Exp → 0</b>	<b>Exp → 1</b>	

# Notice for Example: If-Statement

- The entry **M[else-part, else]** contains two entries, i.e. *the dangling else ambiguity*.
- **Disambiguating rule:** *always prefer the rule that generates the current look-ahead token over any other, and thus the production*

Else-part  $\rightarrow$  else statement

ove

Else-part  $\rightarrow \epsilon$

- With this modification, the above table will become unambiguous
  - The grammar can be parsed **as if** it were an LL(1) grammar

# The parsing based LL(1) Table

- The parsing actions for the string:  
If (0) if (1) other else other
- ( for conciseness, statement=S, if-stmt=I, else-part=L, exp=E, if=I, else=e, other=o)

Steps	Parsing Stack	Input	Action
1	\$S	i(0)i(1)oeo\$	$S \rightarrow I$
2	\$I	i(0)i(1)oeo\$	$I \rightarrow i(E)SL$
3	\$LS)E(i	i(0)i(1)oeo\$	Match
4	\$ LS)E(	(0)i(1)oeo \$	Match
5	\$ LS)E	0)i(1)oeo \$	$E \rightarrow o$
			Match
			Match
			$S \rightarrow I$
			$I \rightarrow i(E)SL$
			Match
			Match
			$E \rightarrow 1$
			Match
			match
			$S \rightarrow o$
			match
			$L \rightarrow eS$
			Match
			$S \rightarrow o$
			match
			$L \rightarrow \epsilon$
22	\$	\$	accept

## 4.2.3 Left Recursion Removal and Left Factoring



# Repetition and Choice Problem

- **Repetition and choice in LL(1) parsing suffer from similar problems** to be those that occur in recursive-descent parsing
  - and for that reason we have **not yet been able to give an LL(1) parsing table for the simple arithmetic expression** grammar of previous sections.
- Solve these problems for **recursive-descent by using EBNF notation**
  - We cannot apply the same ideas to LL(1) parsing;
  - instead, we **must rewrite the grammar within the BNF notation** into a form that the LL(1) parsing algorithm can accept.

# Two standard techniques for Repetition and Choice

- **Left Recursion removal**

$\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$

(in recursive-descent parsing, EBNF:  $\text{exp} \rightarrow \text{term} \{ \text{addop term} \}$ )

- **Left Factoring**

$\text{If-stmt} \rightarrow \text{if ( exp ) statement}$

$\mid \text{if ( exp ) statement else statement}$

(in recursive-descent parsing, EBNF:

$\text{if-stmt} \rightarrow \text{if (exp) statement [else statement]}$ )

# Left Recursion Removal

- **Left recursion is commonly used to make operations left associative, as in the simple expression grammar, where**

$$\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$$

- **Immediate left recursion:**

**The left recursion occurs only within the production of a single non-terminal.**

$$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$$

- **Indirect left recursion:**

**Never occur in actual programming language grammars, but be included for completeness.**

$$A \rightarrow Bb \mid \dots$$

$$B \rightarrow Aa \mid \dots$$

# CASE 1: Simple Immediate Left Recursion

- $A \rightarrow A\alpha \mid \beta$   
Where,  $\alpha$  and  $\beta$  are strings of terminals and non-terminals;  
 $\beta$  does not begin with  $A$ .
- The grammar will generate the strings of the form.  $\beta\alpha^n$
- *We rewrite this grammar rule into two rules:*  
 $A \rightarrow \beta A'$   
To generate  $\beta$  first;  
 $A' \rightarrow \alpha A' \mid \epsilon$   
To generate the repetitions of  $\alpha$ , using right recursion.

# Example

- $\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$
- To rewrite this grammar to remove left recursion, we obtain

$$\text{exp} \rightarrow \text{term exp}'$$

$$\text{exp}' \rightarrow \text{addop term exp}' \mid \varepsilon$$

## CASE2: General Immediate Left Recursion

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

Where none of  $\beta_1, \dots, \beta_m$  begin with  $A$ .

The solution is similar to the simple case:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$$

# Example

- $\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$
- Remove the left recursion as follows:  
     $\text{exp} \rightarrow \text{term exp}'$   
     $\text{exp}' \rightarrow + \text{term exp}' \mid - \text{term exp}' \mid \varepsilon$

# **CASE3: General Left Recursion**

- **Grammars with no  $\epsilon$ -productions and no cycles**

**(1) A cycle is a derivation of at least one step that begins and ends with same non-terminal:**

$$A \Rightarrow \alpha \Rightarrow A$$

**(2) Programming language grammars do have  $\epsilon$ -productions, but usually in very restricted forms.**



# Algorithm for General Left Recursion Removal

For  $i:=1$  to  $m$  do

For  $j:=1$  to  $i-1$  do

Replace each grammar rule choice of the form

$A_i \rightarrow A_j \beta$  by the rule

$A_i \rightarrow \alpha_1 \beta | \alpha_2 \beta | \dots | \alpha_k \beta,$

where  $A_j \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$  is the current rule for  $A_j$ .

Explanation:

- (1) Picking an arbitrary order for all non-terminals, say,  $A_1, \dots, A_m$ ;
- (2) Eliminates all rules of the form  $A_i \rightarrow A_j \gamma$  with  $j \leq i$ ;
- (3) Every step in such a loop would only increase the index, and thus the original index cannot be reached again.

# Example

Consider the following grammar:

$$A \rightarrow Ba \mid Aa \mid c$$
$$B \rightarrow Bb \mid Ab \mid d$$

Where,  $A_1=A$ ,  $A_2=B$  and  $m=2$

(1) When  $i=1$ , the inner loop does not execute, So only to remove the immediate left recursion of  $A$

$$A \rightarrow BaA' \mid c A'$$
$$A' \rightarrow aA' \mid \varepsilon$$
$$B \rightarrow Bb \mid Ab \mid d$$

# Example

(2) when  $i=2$ , the inner loop execute once, with  $j=1$ ; To eliminate the rule  $B \rightarrow Ab$  by replacing  $A$  with it choices

$$A \rightarrow BaA' \mid cA'$$

$$A' \rightarrow aA' \mid \varepsilon$$

$$B \rightarrow Bb \mid BaA'b \mid cAb \mid d$$

(3) We remove the immediate left recursion of  $B$  to obtain

$$A \rightarrow BaA' \mid cA'$$

$$A' \rightarrow aA' \mid \varepsilon$$

$$B \rightarrow |cA'bB' \mid dB'$$

$$B \rightarrow bB' \mid aA'bB' \mid \varepsilon$$

Now, the grammar has no left recursion.

# Notice

- Left recursion removal not changes the language, but
  - Change the grammar and the parse tree
- This change causes a complication for the parser

# Example

**Simple arithmetic expression grammar**

**$\text{expr} \rightarrow \text{expr addop term} \mid \text{term}$**

**$\text{addop} \rightarrow + \mid -$**

**$\text{term} \rightarrow \text{term mulop factor} \mid \text{factor}$**

**$\text{mulop} \rightarrow *$**

**$\text{factor} \rightarrow (\text{expr}) \mid \text{number}$**

**After removal of the left recursion**

**$\text{exp} \rightarrow \text{term exp}'$**

**$\text{exp}' \rightarrow \text{addop term exp}' \mid \epsilon$**

**$\text{addop} \rightarrow + \mid -$**

**$\text{term} \rightarrow \text{factor term}'$**

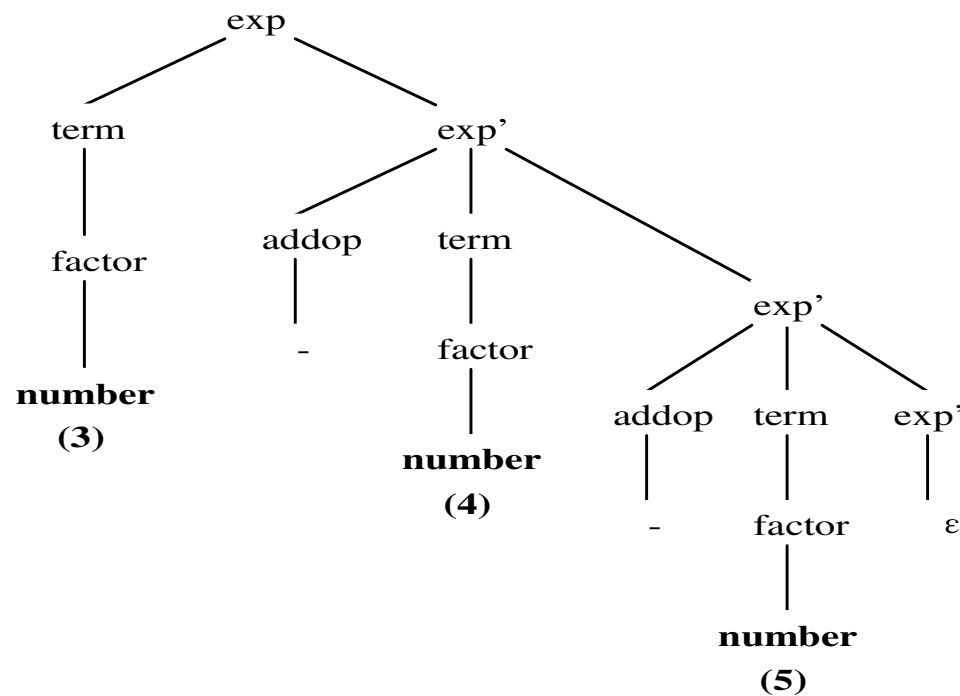
**$\text{term}' \rightarrow \text{mulop factor term}' \mid \epsilon$**

**$\text{mulop} \rightarrow *$**

**$\text{factor} \rightarrow (\text{expr}) \mid \text{number}$**

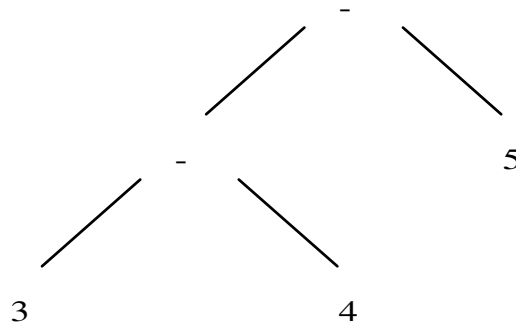
# Parsing Tree

- The parse tree for the expression 3-4-5
  - Not express the left associativity of subtraction.



# Syntax Tree

- Nevertheless, a parse should still construct the appropriate left associative syntax tree



- From the given parse tree, we can see how the value of 3-4-5 is computed.

# Left-Recursion Removed Grammar and its Procedures

- The grammar with its left recursion removed, **exp** and **exp'** as follows:

**exp**  $\rightarrow$  **term exp'**

**exp'**  $\rightarrow$  **addop term exp' |  $\epsilon$**

**Procedure exp**

**Begin**

**Term;**

**Exp';**

**End exp;**

**Procedure exp'**

**Begin**

**Case token of**

**+: match(+);**

**term;**

**exp';**

**-: match(-);**

**term;**

**exp';**

**end case;**

**end exp'**



# Left-Recursion Removed

## Grammar and its Procedures

- To compute the value of the expression,  $\text{exp}'$  needs a parameter from the  $\text{exp}$  procedure

$\text{exp} \rightarrow \text{term exp}'$

$\text{exp}' \rightarrow \text{addop term exp}' \mid \varepsilon$

```
function exp:integer;  
  var temp:integer;  
  Begin  
    Temp:=Term;  
    Return Exp'(temp);  
  End exp;
```

```
function exp'(valsofar:integer):integer;  
  Begin  
    If token=+ or token=- then  
      Case token of  
        +: match(+);  
           valsofar:=valsofar+term;  
        -: match(-);  
           valsofar:=valsofar-term;  
      end case;  
    return exp'(valsofar);
```

# The LL(1) parsing table for the new expression

M[N,T]	(	number	)	+	-	*	\$
Exp	exp → term exp'	exp → term exp'					
Exp'			exp' → ε	exp' → addop term exp'	exp' → addop term exp'		exp' → ε
Addop				addop → +	addop → -		
Term	term → factor term'	term → factor term'					
Term'			term' → ε	term' → ε	term' → ε	term' → mulop factor term'	term' → ε
Mulop						mulop → *	
factor	factor → (expr)	factor → number					

# Left Factoring

- Left factoring is required when two or more grammar rule choices **share a common prefix string**, as in the rule

$$A \rightarrow \alpha \beta | \alpha \gamma$$

- Example:

$$\text{stmt-sequence} \rightarrow \text{stmt}; \text{stmt-sequence} \mid \text{stmt}$$

$$\text{stmt} \rightarrow s$$

- An LL(1) parser **cannot distinguish** between the production choices in such a situation
- The solution in this simple case is to “factor” the  $\alpha$  out on the left and rewrite the rule as two rules:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta | \gamma$$

# Algorithm for Left Factoring a Grammar

**While there are changes to the grammar do**

**For each non-terminal A do**

**Let  $\alpha$  be a prefix of maximal length that is shared**

**By two or more production choices for A**

**If  $\alpha \neq \epsilon$  then**

**Let  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$  be all the production choices for A**

**And suppose that  $\alpha_1, \alpha_2, \dots, \alpha_k$  share  $\alpha$ , so that**

**$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_k | \alpha_{K+1} | \dots | \alpha_n$ , the  $\beta_j$ 's share**

**No common prefix, and  $\alpha_{K+1}, \dots, \alpha_n$  do not share  $\alpha$**

**Replace the rule  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$  by the rules**

**$A \rightarrow \alpha A' | \alpha_{K+1} | \dots | \alpha_n$**

**$A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_k$**

## Example 4.4

- Consider the grammar for statement sequences, written in right recursive form:

$\text{Stmt-sequence} \rightarrow \text{stmt}; \text{stmt-sequence} \mid \text{stmt}$

$\text{Stmt} \rightarrow s$

- Left Factored as follows:

$\text{Stmt-sequence} \rightarrow \text{stmt stmt-seq}'$

$\text{Stmt-seq}' \rightarrow ; \text{stmt-sequence} \mid \varepsilon$

## Example 4.4

- **Notices:**
  - if we had written the stmt-sequence rule left recursively,
  - $\text{Stmt-sequence} \rightarrow \text{stmt-sequence} ; \text{stmt} \mid \text{stmt}$
- **Then removing the immediate left recursion would result in the same rules:**
  - $\text{Stmt-sequence} \rightarrow \text{stmt stmt-seq}'$
  - $\text{Stmt-seq}' \rightarrow ; \text{stmt-sequence} \mid \varepsilon$

## Example 4.5

- Consider the following grammar for if-statements:

**If-stmt  $\rightarrow$  if ( exp ) statement  
                          | if ( exp ) statement else statement**

- The left factored form of this grammar is:

**If-stmt  $\rightarrow$  if (exp) statement else-part  
Else-part  $\rightarrow$  else statement |  $\epsilon$**

## Example 4.6

- An arithmetic expression grammar with right associativity operation:

$$\text{exp} \rightarrow \text{term} + \text{exp} \mid \text{term}$$

- This grammar needs to be left factored, and we obtain the rules

$$\text{exp} \rightarrow \text{term exp}'$$

$$\text{exp}' \rightarrow + \text{exp} \mid \epsilon$$

- Suppose we substitute term exp' for exp, we then obtain:

$$\text{exp} \rightarrow \text{term exp}'$$

$$\text{exp}' \rightarrow + \text{term exp}' \mid \epsilon$$



## Example 4.7

- An typical case where a grammar fails to be LL(1)

Statement  $\rightarrow$  assign-stmt | call-stmt | other

Assign-stmt  $\rightarrow$  identifier := exp

Call-stmt  $\rightarrow$  identifier (exp-list)

- Where, identifier is shared as first token of both assign-stmt and call-stmt and,
- thus, could be the lookahead token for either.
- But not in the form can be left factored.

## Example 4.7

- **First replace assign-stmt and call-stmt by the right-hand sides of their definition productions:**

**Statement  $\rightarrow$  identifier:=exp | identifier(exp-list)  
| other**

- **Then, we left factor to obtain**

**Statement  $\rightarrow$  identifier statement' | other  
Statement'  $\rightarrow$  :=exp |(exp-list)**

- **Note:**
  - **this obscures the semantics of call and assignment by separating the identifier from the actual call or assign action.**

## 4.2.4 Syntax Tree Construction in LL(1) Parsing

# **Difficulty in Construction**

- **It is more difficult for LL(1) to adapt to syntax tree construction than recursive descent parsing**
- **The structure of the syntax tree can be obscured by left factoring and left recursion removal**
- **The parsing stack represents only predicated structure, not structure that have been actually seen**

# Solution

- The solution
  - *Delay the construction of syntax tree nodes to the point when structures are removed from the parsing stack.*
- An extra stack is used to keep track of syntax tree nodes, and
- The “action” markers are placed in the parsing stack to indicate when and what actions on the tree stack should occur

# Example

- A barebones expression grammar with only an addition operation.

$$E \rightarrow E + n \mid n$$

/\* be applied left association\*/

- The corresponding LL(1) grammar with left recursion removal is.

$$E \rightarrow n E'$$

$$E' \rightarrow +nE' \mid \epsilon$$

# To compute the arithmetic value of the expression

- Use a separate stack to store the intermediate values of the computation, called **the value stack**;
  - Schedule two operations on that stack:
    - A push of a number;
    - The addition of two numbers.
    - **PUSH** can be performed **by the match procedure**, and
    - **ADDITION** should be scheduled on the stack, by pushing a **special symbol (such as #)** on the parsing stack.
  - This symbol must also be added to the grammar rule that match a +, namely, the rule for E':
    - $E' \rightarrow +n\#E' | \epsilon$
- **Notes:** The addition is scheduled just after the next number, but before any more E' non-terminals are processed. This guaranteed left associativity.

# The actions of the parser to compute the value of the expression 3+4+5

Parsing Stack	Input	Action	Value Stack
\$E	3+4+5\$	$E \rightarrow n E'$	\$
\$E'n	3+4+5\$	Match/push	\$
\$E'	+4+5\$	$E' \rightarrow +n\#E'$	3\$
\$E'\#n+	+4+5\$	Match	3\$
\$E'\#n	4+5\$	Match/push	3\$
\$E'\#	+5\$	Addstack	43\$
\$E'	+5\$	$E' \rightarrow +n\#E'$	7\$
\$E'\#n+	+5\$	Match	7\$
\$E'\#n	5\$	Match/push	7\$
\$E'\#	\$	Addstack	57\$
\$E'	\$	$E' \rightarrow \epsilon$	12\$
\$	\$	Accept	12\$



## 4.3 First and Follow Sets

The LL(1) parsing algorithm is based on the LL(1) parsing table

The LL(1) parsing table construction involves the First and Follow sets

## 4.3.1 First Sets

# Definition

- Let  $X$  be a grammar symbol( a terminal or non-terminal) or  $\epsilon$ . Then  $\text{First}(X)$  is a set of terminals or  $\epsilon$ , which is defined as follows:
  1. If  $X$  is a terminal or  $\epsilon$ , then  $\text{First}(X) = \{X\}$ ;
  2. If  $X$  is a non-terminal, then for each production choice  $X \rightarrow X_1X_2...X_n$ ,  
 $\text{First}(X)$  contains  $\text{First}(X_1)-\{\epsilon\}$ .  
If also for some  $i < n$ , all the set  $\text{First}(X_1).. \text{First}(X_i)$  contain  $\epsilon$ , the  $\text{first}(X)$  contains  $\text{First}(X_{i+1})-\{\epsilon\}$ .  
IF all the set  $\text{First}(X_1).. \text{First}(X_n)$  contain  $\epsilon$ , the  $\text{First}(X)$  contains  $\epsilon$ .

# Definition

- Let  $\alpha$  be a string of terminals and non-terminals,  $X_1X_2\dots X_n$ .  $\text{First}(\alpha)$  is defined as follows:
  1.  $\text{First}(\alpha)$  contains  $\text{First}(X_1) - \{\epsilon\}$ ;
  2. For each  $i=2,\dots,n$ , if for all  $k=1,\dots,i-1$ ,  $\text{First}(X_k)$  contains  $\epsilon$ , then  $\text{First}(\alpha)$  contains  $\text{First}(X_k) - \{\epsilon\}$ .
  3. IF all the set  $\text{First}(X_1)..\text{First}(X_n)$  contain  $\epsilon$ , the  $\text{First}(\alpha)$  contains  $\epsilon$ .

# Algorithm Computing First (A)

- *Algorithm for computing First(A) for all non-terminal A:*

For all non-terminal A do First(A):={ };

While there are changes to any First(A) do

For each production choice  $A \rightarrow X_1 X_2 \dots X_n$  do

K:=1; Continue:=true;

While Continue= true and  $k \leq n$  do

Add First( $X_k$ )-{ $\epsilon$ } to First(A);

If  $\epsilon$  is not in First( $X_k$ ) then Continue:= false;

k:=k+1;

If Continue = true then add  $\epsilon$  to First(A);

# Algorithm Computing First (A)

- *Simplified algorithm in the absence of  $\varepsilon$ -production.*

For all non-terminal A do First(A):={ };

While there are changes to any First(A) do

For each production choice  $A \rightarrow X_1 X_2 \dots X_n$  do

Add First( $X_1$ ) to First(A);

# About Nullable Non-Terminal

- **Definition:** A non-terminal  $A$  is nullable if there exists a derivation  $A \Rightarrow \epsilon$ .
- **Theorem:** A non-terminal  $A$  is nullable if and only if  $\text{First}(A)$  contains  $\epsilon$ .
- **Proof : 1.** If  $A$  is nullable, then  $\text{First}(A)$  contains  $\epsilon$ .
  - As  $A \Rightarrow^* \epsilon$ , we use induction on the length of a derivation.
  - (1)  $A \Rightarrow \epsilon$ , then there must be a production  $A \rightarrow \epsilon$ , by definition,
  - $\text{First}(A)$  contain  $\text{First}(\epsilon) = \{\epsilon\}$ .
  - (2) Assume the truth of the statement for derivation of length  $< n$ ,
  - and let  $A \Rightarrow X_1 \dots X_K \Rightarrow^* \epsilon$  be a derivation of length  $n$ ;
  - All the  $X_i$  must be non-terminals;
- Implying that each  $X_i \Rightarrow^* \epsilon$ , and in fewer than  $n$  steps.
- Thus, by the induction assumption, for each  $i$   $\text{First}(X_i) = \{\epsilon\}$
- Finally, by definition,  $\text{First}(A)$  must contain  $\epsilon$ .

# Example

- Simple integer expression grammar      Write out each choice separately in order:

$\text{exp} \rightarrow \text{expr addop term}$   
           $\mid \text{term}$   
 $\text{addop} \rightarrow + \mid -$   
 $\text{term} \rightarrow \text{term mulop factor}$   
           $\mid \text{factor}$   
 $\text{mulop} \rightarrow *$   
 $\text{factor} \rightarrow (\text{expr}) \mid \text{number}$

- (1)  $\text{exp} \rightarrow \text{exp addop term}$
- (2)  $\text{exp} \rightarrow \text{term}$
- (3)  $\text{addop} \rightarrow +$
- (4)  $\text{addop} \rightarrow -$
- (5)  $\text{term} \rightarrow \text{term mulop factor}$
- (6)  $\text{term} \rightarrow \text{factor}$
- (7)  $\text{mulop} \rightarrow *$
- (8)  $\text{factor} \rightarrow (\text{exp})$
- (9)  $\text{factor} \rightarrow \text{number}$



# First Set for Above Example

- We can use the simplified algorithm as there exists no  $\epsilon$ -production
- The First sets are as follows:
  - $\text{First}(\text{exp}) = \{ (, \text{number} \}$
  - $\text{First}(\text{term}) = \{ (, \text{number} \}$
  - $\text{First}(\text{factor}) = \{ (, \text{number} \}$
  - $\text{First}(\text{addop}) = \{ +, - \}$
  - $\text{First}(\text{mulop}) = \{ * \}$

## The computation process for above First Set

Grammar Rule	Pass 1	Pass 2	Pass 3
$\text{expr} \rightarrow \text{expr addop term}$			
$\text{expr} \rightarrow \text{term}$			$\text{First}(\text{exp})=\{ (, \text{number} \}$
$\text{addop} \rightarrow +$	$\text{First}(\text{addop})=\{ + \}$		
$\text{addop} \rightarrow -$	$\text{First}(\text{addop})=\{ +, - \}$		
$\text{term} \rightarrow \text{term mulop factor}$			
$\text{term} \rightarrow \text{factor}$		$\text{First}(\text{term})=\{ (, \text{number} \}$	
$\text{mulop} \rightarrow *$	$\text{First}(\text{mulop})=\{ * \}$		
$\text{factor} \rightarrow (\text{expr})$	$\text{First}(\text{factor})=\{ ( \}$		
$\text{factor} \rightarrow \text{number}$	$\text{First}(\text{factor})=\{ (, \text{number} \}$		

# Example

- Left factored grammar of if-statement  
Statement  $\rightarrow$  if-stmt | other  
If-stmt  $\rightarrow$  if (exp) statement else-part  
Else-part  $\rightarrow$  else statement |  $\epsilon$   
Exp  $\rightarrow$  0 | 1
- We write out the grammar rule choice separately and number them:
  - (1) Statement  $\rightarrow$  if-stmt
  - (2) Statement  $\rightarrow$  other
  - (3) If-stmt  $\rightarrow$  if (exp) statement else-part
  - (4) Else-part  $\rightarrow$  else statement
  - (5) Else-part  $\rightarrow \epsilon$
  - (6) Exp  $\rightarrow$  0
  - (7) Exp  $\rightarrow$  1

# The First Set for Above Example

- Note:
  - This grammar does have an  $\varepsilon$ -production, but the only nullable non-terminal *else-part* will not in the beginning of left side of any rule choice and will not complicate the computation process.
- The First Sets:
  - First(statement)={ if,other }
  - First(if-stmt)={ if }
  - First(else-part)={ else, $\varepsilon$  }
  - First(exp)={ 0,1 }

# The computation process for above First Set

Grammar Rule	Pass 1	Pass 2
Statement $\rightarrow$ if-stmt		First(statement)={if,other}
Statement $\rightarrow$ other	First(statement)={other}	
If-stmt $\rightarrow$ if (exp) statement else-part	First(if-stmt)={if}	
Else-part $\rightarrow$ else statement	First(else-part)={else}	
Else-part $\rightarrow \epsilon$	First(else-part)={else, $\epsilon$ }	
Exp $\rightarrow$ 0	First(exp)={1}	
Exp $\rightarrow$ 1	First(exp)={0,1}	

# Example

- Grammar for statement sequences:
  - $\text{Stmt-sequence} \rightarrow \text{stmt stmt-seq}'$
  - $\text{Stmt-seq}' \rightarrow ; \text{stmt-sequence} \epsilon$
  - $\text{stmt} \rightarrow s$
- We list the production choices individually:
  - $\text{Stmt-sequence} \rightarrow \text{stmt stmt-seq}'$
  - $\text{Stmt-seq}' \rightarrow ; \text{stmt-sequence}$
  - $\text{Stmt-seq}' \rightarrow \epsilon$
  - $\text{stmt} \rightarrow s$
- The First sets are as follows:
  - $\text{First}(\text{stmt-sequence}) = \{s\}$
  - $\text{First}(\text{stmt-seq}') = \{;, \epsilon\}$
  - $\text{First}(\text{stmt}) = \{s\}$

## 4.3.2 Follow Sets

# Definition

**Given a non-terminal  $A$ , the set  $\text{Follow}(A)$  is defined as follows.**

- (1) if  $A$  is the start symbol, the  $\$$  is in the  $\text{Follow}(A)$ .**
- (2) if there is a production  $B \rightarrow \alpha A \gamma$  then  $\text{First}(\gamma) - \{\epsilon\}$  is in  $\text{Follow}(A)$ .**
- (3) if there is a production  $B \rightarrow \alpha A \gamma$  such that  $\epsilon$  in  $\text{First}(\gamma)$ , then  $\text{Follow}(A)$  contains  $\text{Follow}(B)$ .**



# Definition

- **Note: The symbol \$ is used to mark the end of the input.**
  - The empty “pseudotoken”  $\epsilon$  is never an element of a follow set.
  - Follow sets are defined only for non-terminal.
  - Follow sets work “on the right” in production while First sets work “on the left” in the production.
- **Given a grammar rule  $A \rightarrow \alpha B$ , Follow(B) will contain Follow(A),**
  - the opposite of the situation for first sets, if  $A \rightarrow B\alpha$ , First(A) contains First(B), except for  $\epsilon$ .

# Algorithm for the computation of follow sets

- $\text{Follow}(\text{start-symbol}) := \{\$ \};$
- For all non-terminals  $A \neq \text{start-symbol}$  do  
 $\text{follow}(A) := \{ \};$
- While there changes to any follow sets do
  - For each production  $A \rightarrow X_1 X_2 \dots X_n$  do
  - For each  $X_i$  that is a non-terminal do
  - Add  $\text{First}(X_{i+1} X_{i+2} \dots X_n) - \{\epsilon\}$  to  $\text{Follow}(X_i)$
  - for  $\epsilon$  is in  $\text{First}(X_{i+1} X_{i+2} \dots X_n)$  then
  - Add  $\text{Follow}(A)$  to  $\text{Follow}(X_i)$

# Example

- The simple expression grammar.
  - (1)  $\text{exp} \rightarrow \text{exp addop term}$
  - (2)  $\text{exp} \rightarrow \text{term}$
  - (3)  $\text{addop} \rightarrow +$
  - (4)  $\text{addop} \rightarrow -$
  - (5)  $\text{term} \rightarrow \text{term mulop factor}$
  - (6)  $\text{term} \rightarrow \text{factor}$
  - (7)  $\text{mulop} \rightarrow *$
  - (8)  $\text{factor} \rightarrow (\text{exp})$
  - (9)  $\text{factor} \rightarrow \text{number}$

# Example

- The first sets:

$\text{First}(\text{exp}) = \{ (, \text{number} \}$

$\text{First}(\text{term}) = \{ (, \text{number} \}$

$\text{First}(\text{factor}) = \{ (, \text{number} \}$

$\text{First}(\text{addop}) = \{ +, - \}$

$\text{First}(\text{mulop}) = \{ * \}$

- The Follow sets:

$\text{Follow}(\text{exp}) = \{ \$, +, -, \} \}$

$\text{Follow}(\text{addop}) = \{ (, \text{number} \}$

$\text{Follow}(\text{term}) = \{ \$, +, -, *, \} \}$

$\text{Follow}(\text{mulop}) = \{ (, \text{number} \}$

$\text{Follow}(\text{factor}) = \{ \$, +, -, *, \} \}$

# The progress of above computation

Grammar rule	Pass 1	Pass 2
$\text{exp} \rightarrow \text{exp addop term}$	$\text{Follow}(\text{exp}) = \{ \$, +, - \}$ $\text{Follow}(\text{addop}) = \{ (, \text{number} )$ $\text{Follow}(\text{term}) = \{ \$, +, - \}$	$\text{Follow}(\text{term}) = \{ \$, +, -, *, ) \}$
$\text{Exp} \rightarrow \text{term}$		
$\text{term} \rightarrow \text{term mulop factor}$	$\text{Follow}(\text{term}) = \{ \$, +, -, * \}$ $\text{Follow}(\text{mulop}) = \{ (, \text{number} )$ $\text{Follow}(\text{factor}) = \{ \$, +, -, * \}$	$\text{Follow}(\text{factor}) = \{ \$, +, -, *, ) \}$
$\text{term} \rightarrow \text{factor}$		
$\text{factor} \rightarrow (\text{exp})$	$\text{Follow}(\text{exp}) = \{ \$, +, -, ) \}$	

# Example

- The simplified grammar of if-statements:
  - (1) Statement  $\rightarrow$  if-stmt
  - (2) Statement  $\rightarrow$  other
  - (3) If-stmt  $\rightarrow$  if (exp) statement else-part
  - (4) Else-part  $\rightarrow$  else statement
  - (5) Else-part  $\rightarrow \epsilon$
  - (6) Exp  $\rightarrow$  0
  - (7) Exp  $\rightarrow$  1

# Example

- The First sets:  
     $\text{First}(\text{statement}) = \{\text{if}, \text{other}\}$   
     $\text{First}(\text{if-stmt}) = \{\text{if}\}$   
     $\text{First}(\text{else-part}) = \{\text{else}, \epsilon\}$   
     $\text{First}(\text{exp}) = \{0, 1\}$
- Computing the following Follow sets:  
     $\text{Follow}(\text{statement}) = \{\$, \text{else}\}$   
     $\text{Follow}(\text{if-statement}) = \{\$, \text{else}\}$   
     $\text{Follow}(\text{else-part}) = \{\$, \text{else}\}$   
     $\text{Follow}(\text{exp}) = \{)\}$

# Example

- The simplified statement sequence grammar.
  - (1)  $\text{Stmt-sequence} \rightarrow \text{Stmt Stmt-seq}'$
  - (2)  $\text{Stmt-seq}' \rightarrow ; \text{Stmt-sequence}$
  - (3)  $\text{Stmt-seq}' \rightarrow \epsilon$
  - (4)  $\text{Stmt} \rightarrow s$



# Example

- The First sets are as follows:

$\text{First}(\text{Stmt-sequence}) = \{s\}$

$\text{First}(\text{Stmt}) = \{s\}$

$\text{First}(\text{Stmt-seq}') = \{;, \epsilon\}$

- And, the Follow sets:

$\text{Follow}(\text{Stmt-sequence}) = \{\$ \}$

$\text{Follow}(\text{Stmt}) = \{;, \$ \}$

$\text{Follow}(\text{Stmt-seq}') = \{\$ \}$

### 4.3.3 Constructing LL(1) Parsing Tables

# The table-constructing rules

- (1) If  $A \rightarrow \alpha$  is a production choice, and there is a derivation  $\alpha \Rightarrow^* a\beta$ , where  $a$  is a token, then add  $A \rightarrow \alpha$  to the table entry  $M[A, a]$
  - (2) If  $A \rightarrow \alpha$  is a production choice, and there are derivations  $\alpha \Rightarrow^* \varepsilon$  and  $S\$ \Rightarrow^* \beta A a \gamma$ , where  $S$  is the start symbol and  $a$  is a token (or  $\$$ ), then add  $A \rightarrow \alpha$  to the table entry  $M[A, a]$
- Clearly, the token  $a$  in the rule (1) is in  $\text{First}(\alpha)$ , and the token  $a$  of the rule (2) is in  $\text{Follow}(A)$ .
  - Thus we can obtain the following algorithmic construction of the LL(1) parsing table:

# Algorithm and Theorem

- Repeat the following two steps for each non-terminal  $A$  and production choice  $A \rightarrow \alpha$ .
  - For each token  $a$  in  $\text{First}(\alpha)$ , add  $A \rightarrow \alpha$  to the entry  $M[A, a]$ .
  - If  $\epsilon$  is in  $\text{First}(\alpha)$ , for each element  $a$  of  $\text{Follow}(A)$  (  $a$  token or  $\$$ ), add  $A \rightarrow \alpha$  to  $M[A, a]$ .
- Theorem: A grammar in BNF is LL(1) if the following conditions are satisfied.
  - For every production  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ ,  $\text{First}(\alpha_i) \cap \text{First}(\alpha_j)$  is empty for all  $i$  and  $j$ ,  $1 \leq i, j \leq n$ ,  $i \neq j$ .
  - For every non-terminal  $A$  such that  $\text{First}(A)$  contains  $\epsilon$ ,  $\text{First}(A) \cap \text{Follow}(A)$  is empty.

# Example

- The simple expression grammar.

$\text{exp} \rightarrow \text{term exp}'$

$\text{exp}' \rightarrow \text{addop term exp}' | \epsilon$

$\text{addop} \rightarrow + -$

$\text{term} \rightarrow \text{factor term}'$

$\text{term}' \rightarrow \text{mulop factor term}' | \epsilon$

$\text{mulop} \rightarrow *$

$\text{factor} \rightarrow (\text{expr}) | \text{number}$

# The first and follow set

First Sets	Follow Sets
First(exp)={ (,number)	Follow(exp)={ \$, ) }
First(exp')={ +, -, $\epsilon$ }	Follow(exp')={ \$, ) }
First(term)={ (,number)	Follow(addop)={ (,number)
First(term')={ *, $\epsilon$ }	Follow(term)={ \$, +, -, ) }
First(factor)={ (,number}	Follow(term')={ \$, +, -, ) }
First(addop)={ +, - }	Follow(mulop)={ (,number}
First(mulop)={ * }	Follow(factor)={ \$, +, -, *, ) }

# the LL(1) parsing table

M[N,T]	(	number	)	+	-	*	\$
Exp	exp $\rightarrow$ term exp'	exp $\rightarrow$ term exp'					
Exp'			exp' $\rightarrow \epsilon$	exp' $\rightarrow$ addop term exp'	exp' $\rightarrow$ addop term exp'		exp' $\rightarrow \epsilon$
Addop				addop $\rightarrow$ +	addop $\rightarrow$ -		
Term	term $\rightarrow$ factor term'	term $\rightarrow$ factor term'					
Term'			term' $\rightarrow$ $\epsilon$	term' $\rightarrow$ $\epsilon$	term' $\rightarrow$ $\epsilon$	term' $\rightarrow$ mulop factor term'	term' $\rightarrow$ $\epsilon$
Mulop						mulop $\rightarrow$ *	
factor	factor $\rightarrow$ (expr)	factor $\rightarrow$ number					

# Example

- The simplified grammar of if-statements

Statement  $\rightarrow$  if-stmt | other

If-stmt  $\rightarrow$  if (exp) statement else-part

Else-part  $\rightarrow$  else statement |  $\varepsilon$

Exp  $\rightarrow$  0 | 1



# The first and follow set

First Sets	Follow Sets
First(statement)={ if,other } First(if-stmt)={ if } First(else-part)={ else, $\epsilon$ } First(exp)={ 0,1 }	Follow(statement)={ \$,else } Follow(if-statement)={ \$,else } Follow(else-part)={ \$,else } Follow(exp)={ ) }

# the LL(1) parsing table

M[N,T]	If	Other	Else	0	1	\$
Statement	Statement $\rightarrow$ if-stmt	Statement $\rightarrow$ other				
If-stmt	If-stmt $\rightarrow$ if (exp) statement else-part					
Else-part			Else-part $\rightarrow$ else statement Else-part $\rightarrow \epsilon$			Else-par t $\rightarrow \epsilon$
exp				Exp $\rightarrow$ 0	Exp $\rightarrow$ 1	

# Example

- Consider the following grammar with left factoring applied.
  - (1)  $\text{Stmt-sequence} \rightarrow \text{Stmt Stmt-seq}'$
  - (2)  $\text{Stmt-seq}' \rightarrow ; \text{Stmt-sequence} \mid \varepsilon$
  - (3)  $\text{Stmt} \rightarrow s$

# The first and follow set

First Sets	Follow Sets
$\text{First}(\text{Stmt-sequence}) = \{s\}$	$\text{Follow}(\text{Stmt-sequence}) = \{\$ \}$
$\text{First}(\text{Stmt}) = \{s\}$	$\text{Follow}(\text{Stmt}) = \{ ; , \$ \}$
$\text{First}(\text{Stmt-seq}') = \{ ; , \epsilon \}$	$\text{Follow}(\text{Stmt-seq}') = \{\$ \}$

# the LL(1) parsing table

M[N,T]	S	;	\$
Stmt-sequence	$\text{Stmt-sequence} \rightarrow$ $\text{Stmt Stmt-seq}'$		
Stmt	$\text{Stmt} \rightarrow s$		
Stmt-seq'		$\text{Stmt-seq}' \rightarrow ;$ $\text{Stmt-sequence}$	$\text{Stmt-seq}' \rightarrow   \epsilon$

## 4.3.4 Extending the lookahead: LL(k) Parsers

# Definition of LL(k)

- The LL(1) parsing method can be extended to k symbols of look-ahead.
- Definitions:
  - $\text{First}_k(\alpha) = \{ wk \mid \alpha \Rightarrow^* w \}$ , where,  $wk$  is the first  $k$  tokens of the string  $w$  if the length of  $w > k$ , otherwise it is the same as  $w$ .
  - $\text{Follow}_k(A) = \{ wk \mid S\$ \Rightarrow^* \alpha A w \}$ , where,  $wk$  is the first  $k$  tokens of the string  $w$  if the length of  $w > k$ , otherwise it is the same as  $w$ .
- LL(k) parsing table:
  - The construction can be performed as that of LL(1).

# Complications in LL(k)

- The complications in LL(k) parsing:
  - The parsing table become larger; since the number of columns increases exponentially with k.
  - The parsing table itself does not express the complete power of LL(k) because the follow strings do not occur in all contexts.
  - Thus parsing using the table as we have constructed it is distinguished from LL(k) parsing by calling it Strong LL(k) parsing, or SLL(k) parsing.



# Complications in LL(k)

- The LL(k) and SLL(k) parsers are uncommon.
  - Partially because of the added complexity;
  - Primarily because of the fact that a grammar fails to be LL(1) is in practice likely not to be LL(k) for any k.

## 4.4 A Recursive-Descent Parser For The Tiny Language

# The Grammar of the TINY language in BNF

- $program \rightarrow stmt\text{-}sequence$
- $stmt\text{-}sequence \rightarrow stmt\text{-}sequence; statement / statement$
- $statement \rightarrow if\text{-}stmt / repeat\text{-}stmt / assign\text{-}stmt / read\text{-}stmt / write\text{-}stmt$
- $if\text{-}stmt \rightarrow \text{if } exp \text{ then } stmt\text{-}sequence \text{ end}$
- $\quad \quad \quad / \text{if } exp \text{ then } stmt\text{-}sequence \text{ else } stmt\text{-}sequence \text{ end}$
- $repeat\text{-}stmt \rightarrow \text{repeat } stmt\text{-}sequence \text{ until } exp$
- $assign\text{-}stmt \rightarrow identifier := exp$
- $read\text{-}stmt \rightarrow \text{read identifier}$
- $write\text{-}stmt \rightarrow \text{write } exp$

# The Grammar of the TINY language in BNF

- $exp \rightarrow simple-exp \text{ comparison-op } simple-exp / simple-exp$
- $comparison-op \rightarrow < | =$
- $simple-exp \rightarrow simple-exp \text{ addop } term / term$
- $addop \rightarrow + | -$
- $term \rightarrow term \text{ mulop } factor \text{ factor } / factor$
- $mulop \rightarrow * | /$
- $factor \rightarrow (exp) \text{ number identifier}$

# TINY PARSER CODES

- The TINY parser consists of two code files:
  - **parse.h and parse.c**
  - The parse.h: (see appendix B, lines 850-865)
  - **TreeNode \* parse(void)**
  - The main routine parse will return a pointer to the syntax tree constructed by the parser.
  - The parse.c(see appendix B, lines 900-1114)
- **11 mutually recursive procedure** that correspond directly to the EBNF grammar.
- **The operators non-terminals** are recognized as part of their associated expressions.

# TINY PARSER CODES

- The static variable **token** is used to keep the look-ahead token
- The contents of each recursive procedures should be relatively **self-explanatory** except stmt-sequence
- The **utility procedures** used by the recursive parsing procedures in util.c:( Appendix B, lines 350-526).
- **NewStmtNode**(line 405-421): take the type of statement as parameter;
- 
- Allocate a new statement node of this kind;
- Return a pointer to the newly allocated node.
- **NewExpNode**(line 423-440): take the type of exp ad parameter;

# TINY PARSER CODES

- Allocate a new exp node of this kind;
- Return a pointer to the new allocated node.
- **Copystring**(line 442-455): take a string as parameter;
- Allocate a sufficient space for a copy, and copy the string;
- Return a pointer to the newly allocated copy.
- A procedure PrintTree in util.c (line 473-506) writes a linear version of the syntax tree to the listing, so that we may view the result of a parse.

End of Part Two

THANKS