Compiler Principle and Technology

Prof. Dongming LU
Mar. 27th , 2015

4. Top-Down Parsing

PART TWO

Contents

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 ε. Then First(X) is a set of terminals or ε, which is defined as follows:

  1. If X is a terminal or ε, then First(X) = {X};
  2. If X is a non-terminal, then for each production choice $X \rightarrow X_1X_2\ldots X_n$,

     First(X) contains First(X_1)-{ε}.

     If also for some i<n, all the set First(X_1)..First(X_i)

     contain ε,the first(X) contains First(X_i+1)-{ε}.

     IF all the set First(X_1)..First(X_n) contain ε, the

     First(X) contains ε.

Definition

Let α be a string of terminals and non-terminals, $X_1X_2 \ldots X_n$. First(α) is defined as follows:

1. First(α) contains First(X_1)-{ε};
2. For each $i=2,\ldots,n$, if for all $k=1,\ldots,i-1$, First(X_k) contains ε, then First(α) contains First(X_k)-{ε}.
3. IF all the set First(X_1)..First(X_n) contain ε, the First(α) contains ε.

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→$X_1X_2\ldots X_n$ do

  K:=1; Continue:=true;

  While Continue= true and k<=n do

  Add First(X_k)-{ε} to First(A);

  If ε is not in First(X_k) then Continue:= false;

  k:=k+1;

  If Continue = true then add ε to First(A);

Algorithm Computing First (A)

- *Simplified algorithm in the absence of ε-production.*

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

    While there are changes to any First(A) do

    For each production choice A→$X_1X_2\ldots X_n$ do

    Add First(X_1) to First(A);

Notice: the First set can include *'ε' symbol, and can't include '$'.*

This is because the First set represents the symbols that can be generated by a non-terminal symbol, and '$' can't be generated by any non-terminal symbol.

Similarly, the Follow set can include *'$' , but not 'ε'*

Example

- Simple integer expression grammar

  exp → exp addop term
  | term
  addop → + | -
  term → term mulop factor
  | factor
  mulop → *
  factor → (expr) | number

Write out each choice separately in order:

(1) exp → exp addop term
(2) exp → term
(3) addop → +
(4) addop → -
(5) term → term mulop factor
(6) term → factor
(7) mulop → *
(8) factor → (exp)
(9) factor → number

First Set for Above Example

- We can use the simplified algorithm as there exists no ε-production
- The First sets are as follows:

  First(exp)={(,number}

  First(term)={(,number}

  First(factor)={(,number}

  First(addop)={+,-}

  First(mulop)={*}

The computation process for above First Set

Grammar Rule	Pass 1	Pass 2	Pass 3
expr → expr addop term			
expr → term			
addop → +	First(addop)={+}		
addop → -	First(addop)={+,-}		
term → term mulop factor			
term → factor			
mulop →*	First(mulop)={*}		
factor →(expr)	First(factor)={(}		
factor →number	First(factor)={(,number)		

The computation process for above First Set

Grammar Rule	Pass 1	Pass 2	Pass 3
expr → expr addop term			
expr → term			
addop → +	First(addop)={+}		
addop → -	First(addop)={+,-}		
term → term mulop factor			
term → factor		First(term)={(,number}	
mulop →*	First(mulop)={*}		
factor →(expr)	First(factor)={(}		
factor →number	First(factor)={(,number)		

The computation process for above First Set

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

Example

- Left factored grammar of if-statement
  Statement → if-stmt | other
  If-stmt → if (exp) statement else-part
  Else-part → else statement | ε
  Exp → 0 | 1

- We write out the grammar rule choice separately and number them:
  (1) Statement → if-stmt
  (2) Statement → other
  (3) If-stmt → if (exp) statement else-part
  (4) Else-part → else statement
  (5) Else-part →ε
  (6) Exp → 0
  (7) Exp → 1

The First Set for Above Example

- **Note:**
  This grammar does have an ε-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.

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

The First Set for Above Example

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

The First Sets:

First(statement)={if,other}
First(if-stmt)={if}
First(else-part)={else,ε}
First(exp)={0,1}

4.3.2 Follow Sets

Definition

Given a non-terminal A, the set Follow(A) is defined as follows.

(1) If A is the start symbol, then \$ is in the Follow(A).

(2) If there is a production $B \to \alpha A \gamma$, then First(γ)-{ε} is in Follow(A).

(3) If there is a production $B \to \alpha A \gamma$ such that ε in First(γ), then Follow(A) contains Follow(B).

Definition

- Note:
  - The symbol $ is used to mark the end of the input.
  - The empty "pseudotoken" ε 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 possibly for ε.

Algorithm for the computation of follow sets

- Follow(start-symbol):={$};
- For all non-terminals A≠start-symbol do follow(A):={ };

While there changes to any follow sets do

For each production $A \rightarrow X_1 X_2 \ldots X_n$ do

For each X_i that is a non-terminal do

Add First(X_i+1X_i+2…X_n) – {ε} to Follow(X_i)

if ε is in First(X_i+1X_i+2…X_n) then

Add Follow(A) to Follow(X_i)

Example

- The simple expression grammar.
  - (1) exp → exp addop term
  - (2) exp → term
  - (3) addop → +
  - (4) addop → -
  - (5) term → term mulop factor
  - (6) term → factor
  - (7) mulop →*
  - (8) factor →(exp)
  - (9) factor →number

The first sets:

First(exp)={(,number}

First(term)={(,number}

First(factor)={(,number}

First(addop)={+,-}

First(mulop)={*}

Compute Follow sets

The simple expression grammar.
(1) exp → exp addop term
(2) exp → term
(3) addop → + (4) addop → -
(5) term → term mulop factor
(6) term → factor
(7) mulop →*
(8) factor →(exp)
(9) factor →number

The first sets:
First(exp)={(,number}
First(term)={(,number}
First(factor)={(,number}
First(addop)={+,-}
First(mulop)={*}

Grammar rule	Pass 1	Pass 2
exp → exp addop term	**Follow(exp)={$,+,- }**	
exp → term		
term → term mulop factor		
term →factor		
factor →(exp)		

Compute Follow sets

The simple expression grammar.
(1) exp → exp addop term
(2) exp → term
(3) addop → + (4) addop → -
(5) term → term mulop factor
(6) term → factor
(7) mulop →*
(8) factor →(exp)
(9) factor →number

The first sets:
First(exp)={(,number}
First(term)={(,number}
First(factor)={(,number}
First(addop)={+,-}
First(mulop)={*}

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

Compute Follow sets

The simple expression grammar.
(1) exp → exp addop term
(2) exp → term
(3) addop → + (4) addop → -
(5) term → term mulop factor
(6) term → factor
(7) mulop →*
(8) factor →(exp)
(9) factor →number

The first sets:
First(exp)={(,number}
First(term)={(,number}
First(factor)={(,number}
First(addop)={+,-}
First(mulop)={*}

Grammar rule	Pass 1	Pass 2
exp → exp addop term	Follow(exp)={$,+,- } Follow(addop)={(,number)	
exp → term		
term → term mulop factor	**Follow(term)=** **Follow(exp) U First(mulop)** **={ $,+,-, *}**	
term →factor		
factor →(exp)		

Compute Follow sets

The simple expression grammar.
(1) exp → exp addop term
(2) exp → term
(3) addop → + (4) addop → -
(5) term → term mulop factor
(6) term → factor
(7) mulop →*
(8) factor →(exp)
(9) factor →number

The first sets:
First(exp)={(,number}
First(term)={(,number}
First(factor)={(,number}
First(addop)={+,-}
First(mulop)={*}

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

Compute Follow sets

The simple expression grammar.
(1) exp → exp addop term
(2) exp → term
(3) addop → + (4) addop → -
(5) term → term mulop factor
(6) term → factor
(7) mulop →*
(8) factor →(exp)
(9) factor →number

The first sets:
First(exp)={(,number}

First(term)={(,number}

First(factor)={(,number}
First(addop)={+,-}
First(mulop)={*}

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

Compute Follow sets

The Follow sets:

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

Follow(addop)={(,number)

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

Follow(mulop)={(,number)

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

Grammar rule	Pass 1	Pass 2
exp → exp addop term	Follow(exp)={$,+,- } Follow(addop)={(,number) Follow(term)={ $,+,-}	Follow(term)={ $,+,-, *,) }
Exp → term		
term → term mulop factor	Follow(term)={ $,+,-, *} Follow(mulop)={(,number) Follow(factor)={ $,+,-, *}	Follow(factor)={ $,+,-, *,) } **add ')'**
term → factor		
factor → (exp)	Follow(exp)={$,+,-,) }	

The LL(1) Grammar

Theorem: A grammar in BNF is LL(1) if the following conditions are satisfied.

- For every production $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$,

  $$\mathbf{First}(\alpha_i) \cap \mathbf{First}(\alpha_j) = \varnothing,$$

  for all i and j, $1 \leqq i,j \leqq n$, $i \neq j$.

- For every non-terminal A such that First(A) contains $\boldsymbol{\varepsilon}$,

  $$\mathbf{First}(A) \cap \mathbf{Follow}(A) = \varnothing$$

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 First(α), and the token a of the rule (2) is in Follow(A),thus we can obtain the following algorithmic construction of the LL(1) parsing table:

Repeat the following two steps for each non-terminal A and production choice ***A→α***.

- For each token a in First(α), add ***A→α*** to the entry M[A,a].
- If ε is in First(α), for each element a of Follow(A) (a token or \$), add ***A→α*** to M[A,a].

Example

- The simple expression grammar.
  exp → term exp'
  exp'→ addop term exp'|ε
  addop → + -
  term → factor term'
  term' → mulop factor term'|ε
  mulop →*
  factor →(expr) | number

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

The LL(1) parsing table

Repeat the following two steps for each non-terminal A and production choice ***A→α***.

- **For each token *a* in First(α), add *A→α* to the entry M[A,*a*].**
- If ε is in First(α), for each element *a* of Follow(A) (a token or $), add ***A→α*** to M[A,*a*].

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

The LL(1) parsing table

Repeat the following two steps for each non-terminal A and production choice ***A→α***.

- ➢ For each token *a* in First(α), add ***A→α*** to the entry M[A,*a*].
- ➢ **If ε is in First(α), for each element *a* of Follow(A) (a token or $), add *A→α* to M[A,*a*].**

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					

Example

- The simplified grammar of if-statements

  Statement → if-stmt | other

  If-stmt → if (exp) statement else-part

  Else-part → else statement | ε

  Exp → 0 | 1

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

The LL(1) parsing table

M[N,T]	If	Other	Else	0	1	$
Statement	Statement → if-stmt	Statement → other				
If-stmt	If-stmt → if (exp) statement else-part					
Else-part			Else-part → else statement Else-part → ε			Else-part → ε
exp				Exp → 0	Exp → 1	

4.3.4 Extending the lookahead: LL(k) Parsers

Definition of LL(k)

- The LL(1) parsing method can be extend to k symbols of look-ahead.
- Definitions:
  - ➢ Firstk(α)={*w*k | α=>* *w*}, where, *w*k is the first k tokens of the string *w* if the length of w > k, otherwise it is the same as *w*.
  - ➢ Followk(A)={*w*k | S$=>*αA*w*}, where, *w*k 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.

- The LL(k) and SLL(k) parsers are uncommon.

Partially because of the added complex. 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.

LL(1) can express the most common case

4.5 Error Recovery in Top-Down Parsers

Basic principles for error recovery in parsers

- RECOGNIZER:

  Only determine if a program is syntactically correct or not. (MINIMUM)

- At a minimum, any parser must behave like a recognizer—that is,
  - ➢ if a program contains a syntax error, the parser must indicate that *some* error exists, and conversely,
  - ➢ if a program contains no syntax errors, then the parser should not claim that an error exists.
- The response of a parser to syntax errors is often a critical factor in the usefulness of a compiler.

Levels of Response

- Beyond this minimal behavior, a parser can be exhibit different levels of response to errors
  - ➢ Give a meaningful error message
  - ➢ Some form of error correction
  - ➢ Missing punctuation, minimal distance error correction

- Most of the techniques for error recovery are ad hoc, and general principles are hard to come by

Goals for error recovery in parsers

- Some important considerations applied:
  - To determine that an error has occurred as soon as possible
  - To parse as much of the code as possible so as to find as many real errors as possible during a single translation
  - To avoid the error cascade problem
  - To avoid infinite loops on error
- Some goals conflict with each other

  Compiler writer is forced to make trade-offs during the construction of an error handler.

4.5.1 Error Recovery in Recursive-Descent Parsers

- Panic mode:
  - ➢ A standard form of error recovery in recursive-decent parsers
  - ➢ 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:
  - ➢ A set of synchronizing tokens are provided to each recursive procedure
  - ➢ If an error is encountered, the parser scans ahead, throwing away tokens until one of the synchronized tokens is seen in the input, whence parsing is resumed.

- The important decisions in this error recovery method:
  - ➢ What tokens to add to the synchronizing set at each point in the parse?
  - ➢ Follow sets are important candidates for such synchronizing tokens.
  - ➢ First sets are used to prevent the error handler from skipping important tokens that begin major new constructs; and also are important to detect errors early in the parse.
- Panic mode works best when the compiler knows when not to panic
  - ➢ Missing punctuation symbols should not always cause an error handler to consume tokens.

Panic Mode Error Recovery in Pseudo-Code

- In addition to the *match* and *error* procedures, we have more procedure
- **Check-input procedure**: performs the early look-ahead checking

```
Procedure Checkinput(Firstset, Followset)
  Begin
  If not (token in firstset) then
        Error;
        Scanto(firstset∪followset)
        End if;
  end
```

Panic Mode Error Recovery in Pseudo-Code

- In addition to the *match* and *error* procedures, we have more procedures.

- **Scanto procedure**: is the panic mode token consumer proper.

```
Procedure scanto(synchset)
  Begin
     While not (token in synchset∪{$}) do
          Gettoken;
 End scanto
```

The exp and factor procedure with panic mode error recovery

```
Procedure exp(synchset)
Begin
        Checkinput({(,number},synchset)
        If not (token in synchset) then
                Term(synchset);
                While token=+ or token=- do
                        Match(token);
                        Term(synchset);
                End while
                Checkinput(synchset,{(,number)};
        End if;
End exp;
```

Notes

- Note: ***Checkinput*** is called twice in each procedure,
  - ➢Once to verify that a token in the First set in the next token in the input
  - ➢a second time to verify that a token in the Follow set is the next token on exit

Example

- This form of panic mode will generate reasonable errors.
- For example: (2+-3)*4-+5 will generate exactly two error messages,
  (1) one at the first minus sign, and
  (2) one at the second plus sign.

Note

- In general, ***synchset*** is to be passed down in the recursive calls, with new synchronizing tokens added as appropriate.
- As a exception, in the case of factor, ***exp*** is called with right parenthesis only as its follow set (*synchset* is discarded)
- The kind of ad hoc analysis accompanies panic mode error recovery.

  For example, (2+*) would not generate a spurious error message at the right parenthesis.

4.5.2 Error Recovery in LL(1) Parsers

- Panic mode error recovery can be implemented in LL(1) parsers in a similar manner to the way it is implemented in recursive-decent parsing
  - ➢ A new stack is required to keep the ***synchset*** parameters;
  - ➢ A call to checkinput must be scheduled by the algorithm before each generate action of the algorithm;
- The primary error situation occurs when
  - ➢ The current input token is not in First(A)

    (or Follow(A), if ε is in First(A))
  - ➢ A is the non-terminal at the top of the stack..
- The case where a token at the top of the stack is not the same as the current input token, does not normally occur

- An alternative to the use of an extra stack
  - ➢ Statically build the sets of synchronizing tokens directly into the LL(1) parsing table,
  - ➢ together with the corresponding actions that ***checkinput*** would take
- Given a non-terminal A at the top of the stack and an input token that is not in First(A)(or Follow(A), if ***ε*** is in First(A)), there are three possible alternative:
  1. Pop A from the stack (by the notation pop, is equivalent to a reduction by an ε-production)
  2. Successively pop tokens from the input until a token is seen for which we can restart the parse (by the notation scan)
  3. Push a new non-terminal onto the stack

Basic Methods

- Choosing alternative 1 if the current input token is \$ or is in Follow(A);

  (indicated by the notation *pop*)
- Alternative 2 if the current input token is not \$ and is not in First(A)∪Follow(A);

  (indicated by the notation *scan*)
- Option 3 is occasionally useful in special situation.

Notice: The rationale of the first choice is that when we choose the 'pop' notation, it is similar to reduce a non-terminal if there was no error, and we usually reduce a non-terminal when we see the Follow set of a non-terminal.

Similarly, we can know the reason of alternative 2 and 3

Example: The Simple Expression Grammar

exp → term exp'
exp'→ addop term exp'|ε
addop → + -
term → factor term'
term' → mulop factor term'|ε
mulop →*
factor →(exp) | number

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

The LL(1) Table

M[N,T]	(	number	)	+	-	*	$
exp	exp → term exp’	exp → term exp’	**pop**	**scan**	**scan**	**scan**	**pop**
exp’	**scan**	**scan**	exp’→ε	exp’→ add op term exp’	exp’→ add op term exp’	**scan**	exp’→ε
addop	**pop**	**pop**	**scan**	addop → +	addop → -	**scan**	**pop**
term	term → factor term’	term → factor term’	**pop**	**pop**	**pop**	**scan**	**pop**
term’	**scan**	**scan**	term’→ε	term’ →ε	term’ →ε	term’ → mulop factor term’	term’ →ε
mulop	**pop**	**pop**	**scan**	**scan**	**scan**	mulop →*	**pop**
factor	factor →(expr)	factor → number	**pop**	**pop**	**pop**	**pop**	**pop**

Example

Given the string (2+*), the prefix (2+ has already been successfully matched)

Parsing stack	Input	Action
$E'T')E'T	*)$	Scan(error)
$E'T')E'T	)$	Pop(error)
$E'T')E'	)$	E'→ε
$E'T')	)$	Match
$E'T'	$	T' →ε
$E'	$	E' →ε
$	$	accept

- Note: There are two adjacent error moves before the parse resumes successfully.
- We can arrange to suppress an error message on the second error move by requiring, after the first error, that parser make one or more successful moves before generating any new error messages. Thus, error message cascades would be avoided.
- There is (at least) one problem in this error recovery method that calls for special action. Since many error actions pop a nonterminal from the stack, it is possible for the stack to become empty, with some input still to parse.

  A simple case of this in the example just given is any string beginning with a right parenthesis: this will cause *E* to be immediately popped, leaving the stack empty with all the input yet to be consumed.
- One possible action that the parser can take in this situation is to push the start symbol on the stack and scan forward the input until a symbol in the First set of the start symbol is seen.

4.5.3 Error Recovery in the TINY Parser

- The error handling of the TINY parser(Given in Appendix B)
- Extremely rudimentary:
  - ➢ only a very primitive form of panic mode recovery is implemented without synchronizing sets
  - ➢ Match procedure: declares error and stating which token is unexpected;
  - ➢ Statement procedure:
  - ➢ Factor procedure: declares error when no correct choice is found.
  - ➢ Parse procedure: declares error if a token other than end of file is found after parse finishes.
- The principle error message generated is "unexpected token"
  - Very unhelpful to the user;
  - Makes no attempt to avoid error cascades.

The sample program with a semicolon added after the write-statement:

```
…
5: read x;
6: if 0< x then
7:        fact :=1;
8:        repeat
9:               fact:=fact *x;
10:              x:x-1
11:       until x =0;
12:       write fact;(<——BAD SEMICOLON;)
13: end
14:
```

Cause the following two error messages to be generated:

1. Syntax error at line 13: unexpected token->reserved word: end
2. Syntax error at lint 14: unexpected token->EOF

The same program with the comparison < deleted in the second line of code

```
…
5: read x;
6: if 0 x then (<——COMPARISON MISSING HERE!;)
7:          fact :=1;
8:          repeat
9:                    fact:=fact *x;
10:                   x:x-1
11:         until x =0;
12:         write fact
13: end
14:
```

cause four error messages to be printed in the listing:

1. Syntax error at line 6: unexpected token->ID, name =x
2. Syntax error at lint 6: unexpected token-> reserved word: then
3. Syntax error at line 6: unexpected token->reserved word: then
4. Syntax error at lint 7: unexpected token->ID, name=fact

- On the other hand, some of the TINY parse's behavior is reasonable.
- For example:
  - A missing semicolon will generate only one error message;
  - The parser will go on to build the correct syntax tree as if the semicolon had been there all along.
- This behavior results from two coding facts:
  - First, the match procedure does not consume a token;
  - Second, the stmt-sequence will connect up as much of the syntax tree as possible in the case of error.

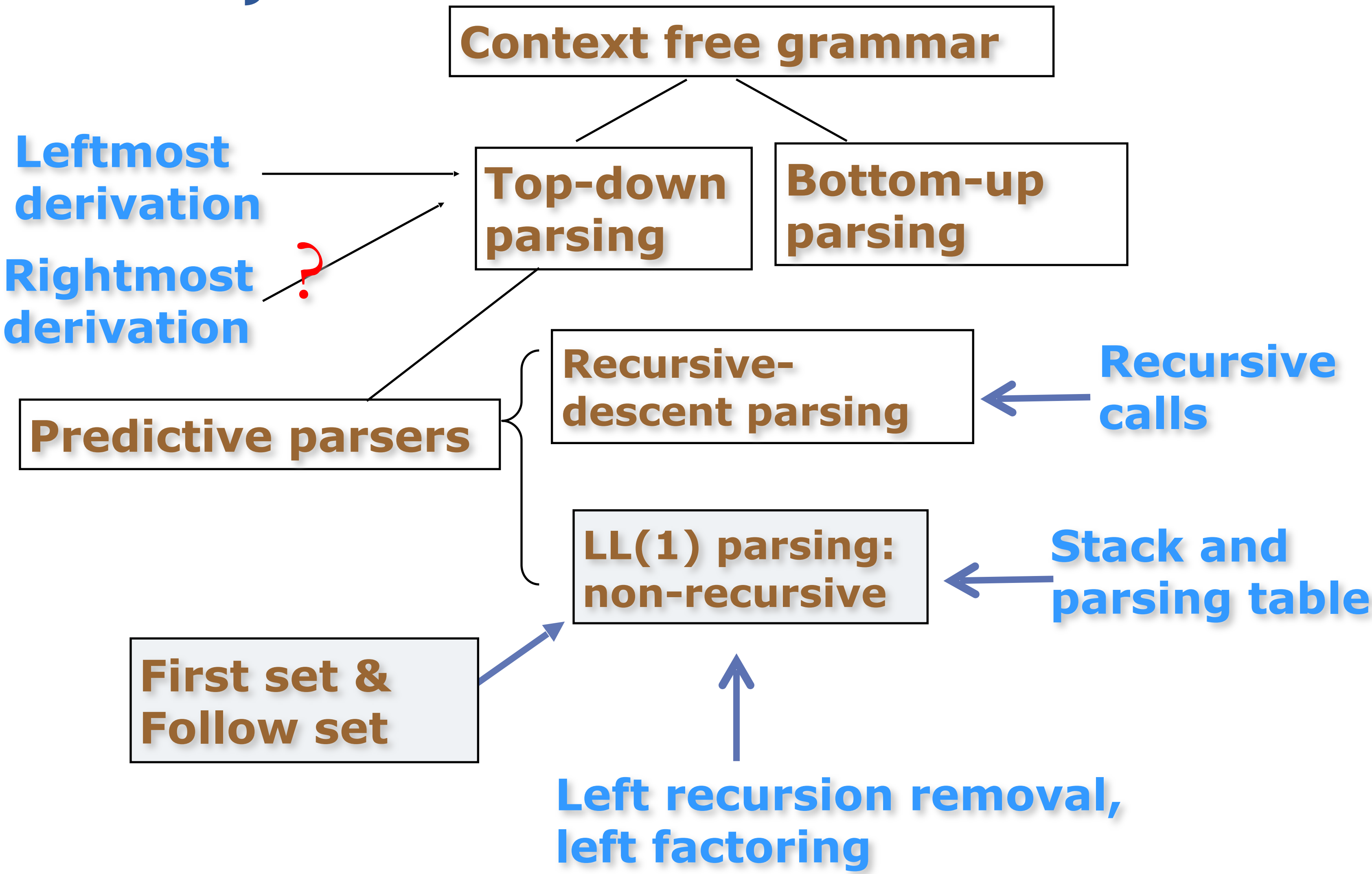
The obvious way of writing the body of stmt-sequence based on the EBNF:

```
statement( );
while (token==SEMI)
{ match (SEMI);
 statement( );
}
```

and the form written with a more complicated loop test:

```
statement( );
while ((token!=ENDFILE) && (token!=END)
 && (token!=ELSE)&&(token!=UNTIL))
{ match (SEMI);
 statement( );
}
```

Summary


Context free grammar
Leftmost derivation
Rightmost derivation
?
Top-down parsing
Bottom-up parsing
Recursive-descent parsing
Recursive calls
Predictive parsers
LL(1) parsing: non-recursive
Stack and parsing table
First set & Follow set
Left recursion removal, left factoring

End of Part Two

THANKS