Chapter 4
Lexical and Syntax Analysis

長庚大學資訊工程學系 陳仁暉 助理教授
Tel: (03) 211-8800 Ext: 5990
E-mail: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/jhchen

ISBN 0-321-33025-0

Chapter 4 Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- Recursive-Descent Parsing
- Bottom-Up Parsing

Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Using BNF to Describe Syntax

- Provides a clear and concise syntax description
- The parser can be based directly on the BNF
- Parsers based on BNF are easy to maintain

Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* – less complex approaches can be used for lexical analysis; separating them simplifies the parser
- *Efficiency* – separation allows optimization of the lexical analyzer
- *Portability* – parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a "front-end" for the parser
- Identifies substrings of the source program that belong together – *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*
  - `sum` is a lexeme; its token may be `IDENT`

Lexical Analysis (continued)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description
  - Design a state diagram that describes the tokens and write a program that implements the state diagram
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram

State Diagram Design

- A naïve state diagram would have a transition from every state on every character in the source language – such a diagram would be very large!

Lexical Analysis (cont.)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent – use a digit class

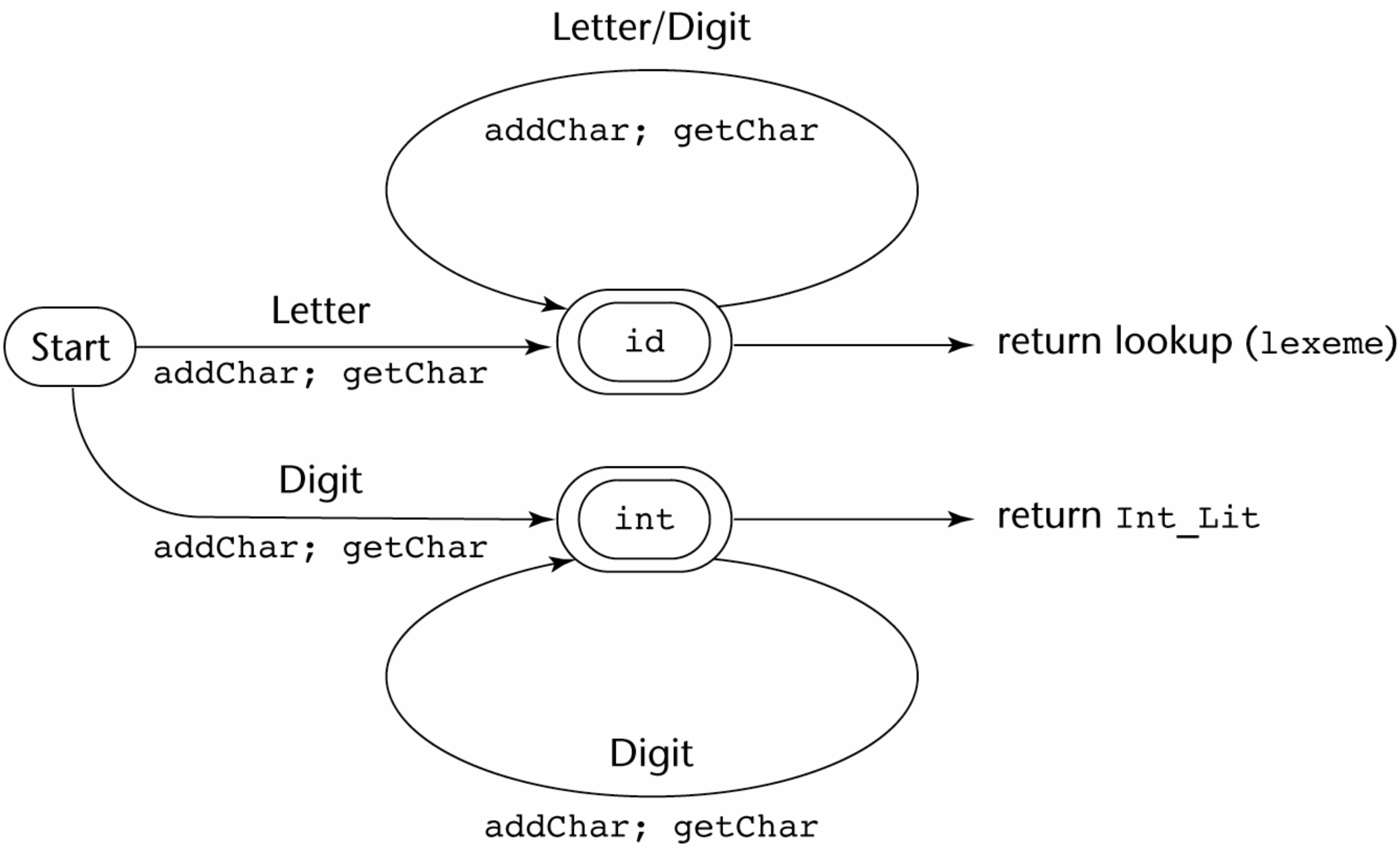
Lexical Analysis (cont.)

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis (cont.)

- Convenient utility subprograms:
  - **getChar** – gets the next character of input, puts it in **nextChar**, determines its class and puts the class in **charClass**
  - **addChar** – puts the character from **nextChar** into the place the lexeme is being accumulated, **lexeme**
  - lookup – determines whether the string in **lexeme** is a reserved word (returns a code)

State Diagram



Lexical Analysis (cont.)

Implementation (assume initialization):

```
int lex() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT)
      {
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;
       ...
```

Lexical Analysis (cont.)

```
...
case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
      break;
  }  /* End of switch */
}  /* End of function lex */
```

The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message, and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

The Parsing Problem (cont.)

- Two categories of parsers
  - *Top down* – produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* – produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Parsers look only one token ahead in the input

The Parsing Problem (cont.)

- Top-down Parsers
  - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
  - Recursive descent – a coded implementation
  - LL parsers – table driven implementation
  - LL means `Left-to-right Leftmost derivation’

The Parsing Problem (cont.)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - The most common bottom-up parsing algorithms are in the LR family
  - LR stands for `Left-to-right Rightmost derivation'

The Parsing Problem (cont.)

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ($O(n)$, where n is the length of the input)

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing (cont.)

- A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | ( <expr> )
```

Recursive-Descent Parsing (cont.)

- Assume we have a lexical analyzer named **`lex`**, which puts the next token code in `nextToken`
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing (cont.)

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

/* Parse the first term */

 term();
  ...
```

Recursive-Descent Parsing (cont.)

```
/* As long as the next token is + or -, call
   lex to get the next token, and parse the
   next term */

  while (nextToken == PLUS_CODE ||
         nextToken == MINUS_CODE){
    lex();
    term();
  }
}
```

- This particular routine does not detect errors
- Convention: Every parsing routine leaves the next token in `nextToken`

Recursive-Descent Parsing (cont.)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive-Descent Parsing (cont.)

```
/* Function factor
   Parses strings in the language
   generated by the rule:
   <factor> -> id  |  (<expr>)  */

 void factor() {

 /* Determine which RHS */

  if (nextToken) == ID_CODE)

 /* For the RHS id, just call lex */

    lex();
```

Recursive-Descent Parsing (cont.)

```
/* If the RHS is (<expr>) - call lex to pass
     over the left parenthesis, call expr, and
     check for the right parenthesis */

  else if (nextToken == LEFT_PAREN_CODE) {
   lex();
     expr();
    if (nextToken == RIGHT_PAREN_CODE)
       lex();
     else
       error();
   }  /* End of else if (nextToken == ...  */

   else error(); /* Neither RHS matches */
 }
```

Recursive-Descent Parsing (cont.)

- The LL Grammar Class
  - The Left Recursion Problem
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
      - A grammar can be modified to remove left recursion

Recursive-Descent Parsing (cont.)

- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness
  - The inability to determine the correct RHS on the basis of one token of lookahead
  - Def: FIRST(α) = $\{a \mid \alpha =>^* a\beta \}$

    (If $\alpha =>^* \varepsilon$, ε is in FIRST(α))

Recursive-Descent Parsing (cont.)

- Pairwise Disjointness Test:
  - For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$$

- Examples:

$$A \rightarrow a \mid bB \mid cAb$$

$$A \rightarrow a \mid aB$$

Recursive-Descent Parsing (cont.)

- Left factoring can resolve the problem

  Replace

  <variable> $\rightarrow$ identifier | identifier [<expression>]

  with

  <variable> $\rightarrow$ identifier <new>

  <new> $\rightarrow$ ε | [<expression>]

  or

  <variable> $\rightarrow$ identifier [[<expression>]]

  (the outer brackets are metasymbols of EBNF)

FIRST Sets

- FIRST(α) is the set of all terminal symbols that can begin some sentential form that starts with α
- FIRST(α) = $\{a \text{ in } V_t \mid \alpha \Rightarrow^* a\beta \} \cup \{ \varepsilon \}$ if $\alpha \Rightarrow^* \varepsilon$
- Example:

  <stmt> → simple | begin <stmts> end

  FIRST(<stmt>) = {simple, begin}

Computing FIRST sets

Initially FIRST(A) is empty

1. For productions $A \rightarrow a\ \beta$, where a in V_t

   Add { a } to FIRST(A)

2. For productions $A \rightarrow \varepsilon$

   Add { ε } to FIRST(A)

3. For productions $A \rightarrow \alpha\ B\ \beta$, where $\alpha \Rightarrow^* \varepsilon$ and NOT ($B \Rightarrow \varepsilon$)

   Add FIRST(αB) to FIRST(A)

4. For productions $A \rightarrow \alpha$, where $\alpha \Rightarrow^* \varepsilon$

   Add FIRST(α) and { ε } to FIRST(A)

To compute FIRST across strings of terminals and non-terminals:

$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$

$\text{FIRST}(A\alpha) = A$ if A is a terminal

$= \text{FIRST}(A) \cup \text{FIRST}(\alpha)$

if $A \rightarrow \varepsilon$

$= \text{FIRST}(A)$ otherwise

Example 1

- S → a S e
- S → B
- B → b B e
- B → C
- C → c C e
- C → d

- FIRST(C) =
- FIRST(B) =
- FIRST(S) =

Example 1

- $S \rightarrow a\ S\ e$
- $S \rightarrow B$
- $B \rightarrow b\ B\ e$
- $B \rightarrow C$
- $C \rightarrow c\ C\ e$
- $C \rightarrow d$

- FIRST(C) = {c,d}
- FIRST(B) = {b,c,d}
- FIRST(S) = {a,b,c,d}

Example 2

- $P \rightarrow i \mid c \mid n\ T\ S$
- $Q \rightarrow P \mid a\ S \mid b\ S\ c\ S\ T$
- $R \rightarrow b \mid \varepsilon$
- $S \rightarrow c \mid R\ n \mid \varepsilon$
- $T \rightarrow R\ S\ q$

- FIRST(P) =
- FIRST(Q) =
- FIRST(R) =
- FIRST(S) =
- FIRST(T) =

Example 2

- $P \rightarrow i \mid c \mid n\ T\ S$
- $Q \rightarrow P \mid a\ S \mid b\ S\ c\ S\ T$
- $R \rightarrow b \mid \varepsilon$
- $S \rightarrow c \mid R\ n \mid \varepsilon$
- $T \rightarrow R\ S\ q$

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,b}
- FIRST(R) = {b, ε}
- FIRST(S) = {c,b,n, ε}
- FIRST(T) = {b,c,n,q}

Example 3

- $S \rightarrow a\ S\ e \mid S\ T\ S$
- $T \rightarrow R\ S\ e \mid Q$
- $R \rightarrow r\ S\ r \mid \varepsilon$
- $Q \rightarrow S\ T \mid \varepsilon$

- FIRST(S) =
- FIRST(R) =
- FIRST(T) =
- FIRST(Q) =

Example 3

- $S \rightarrow a\ S\ e \mid S\ T\ S$
- $T \rightarrow R\ S\ e \mid Q$
- $R \rightarrow r\ S\ r \mid \varepsilon$
- $Q \rightarrow S\ T \mid \varepsilon$

- $\text{FIRST}(S) = \{a\}$
- $\text{FIRST}(R) = \{r, \varepsilon\}$
- $\text{FIRST}(T) = \{r, a, \varepsilon\}$
- $\text{FIRST}(Q) = \{a, \varepsilon\}$

FOLLOW Sets

- FOLLOW(A) is the set of terminals (including end of file) that may follow non-terminal A in some sentential form.
- FOLLOW(A) = {a in V_t | S $\Rightarrow^+$ ...Aa...} U {$ (end of file)} if S $\Rightarrow^+$ ...A
- For example, consider L $\Rightarrow^+$ (())(L)L --

  Both ')' and end of file can follow L

Computing FOLLOW(A)

1. If S is a start symbol, put \$ in FOLLOW(S)
2. Productions of the form $B \rightarrow \alpha\ A\ a$, then add { a } to FOLLOW(A)
3. Productions of the form $B \rightarrow \alpha\ A\ \beta$,

   Add FIRST(β) – {ε} to FOLLOW(A)

   INTUITION: Suppose $B \rightarrow AX$ and FIRST(X) = {c}

   $S \Rightarrow^{+} \alpha\ B\ \beta \Rightarrow \alpha\ A\ X\ \beta \Rightarrow^{+} \alpha\ A\ c\ \delta\ \beta$

4. Productions of the form $B \rightarrow \alpha A$ or $B \rightarrow \alpha A \beta$ where $\beta \Rightarrow^* \varepsilon$

Add FOLLOW(B) to FOLLOW(A)

INTUITION:

– Suppose $B \rightarrow Y A$

$S \Rightarrow^+ \alpha B \beta \Rightarrow \alpha Y A \beta$

– Suppose $B \rightarrow A X$ and $X \rightarrow \varepsilon$

$S \Rightarrow^+ \alpha B \beta \Rightarrow \alpha A X \beta \Rightarrow \alpha A \beta$

NOTE: ε *never* in FOLLOW sets

Example 4

- $S \rightarrow a\ S\ e \mid B$
- $B \rightarrow b\ B\ C\ f \mid C$
- $C \rightarrow c\ C\ g \mid d \mid \varepsilon$

- FIRST(C) = {c,d, ε}
- FIRST(B) = {b,c,d, ε}
- FIRST(S) = {a,b,c,d, ε}

- FOLLOW(C) =
- FOLLOW(B) =
- FOLLOW(S) =

Example 4

- $S \rightarrow a\ S\ e \mid B$
- $B \rightarrow b\ B\ C\ f \mid C$
- $C \rightarrow c\ C\ g \mid d \mid \varepsilon$

- FIRST(C) = {c,d, ε}
- FIRST(B) = {b,c,d, ε}
- FIRST(S) = {a,b,c,d, ε}

- FOLLOW(C) = g,f
  FOLLOW(C) = {c,d,e,f,g,$}
- FOLLOW(B) = c,d,f
  FOLLOW(B) = {c,d,f,$,e}
- FOLLOW(S) = {$, e }

Example 5

- $S \rightarrow (A) \mid \varepsilon$
- $A \rightarrow T\ E$
- $E \rightarrow ,\ T\ E \mid \varepsilon$
- $T \rightarrow (A) \mid a \mid b \mid c$

- FIRST(T) = {(,a,b,c}
- FIRST(E) = {‘,’, ε }
- FIRST(A) = {(,a,b,c}
- FIRST(S) = {(, ε}

- FOLLOW(S) =
- FOLLOW(A) =
- FOLLOW(E) =
- FOLLOW(T) =

Example 5

- S → (A) | ε
- A → T E
- E → , T E | ε
- T → (A) | a | b | c

- FIRST(T) = {(,a,b,c}
- FIRST(E) = {‘,’, ε }
- FIRST(A) = {(,a,b,c}
- FIRST(S) = {(, ε}

- FOLLOW(S) = {$}
- FOLLOW(A) = {) }
- FOLLOW(E) = {) }
- FOLLOW(T) = {‘,’,)}

Example 6

- $E \rightarrow T\ E'$
- $E' \rightarrow +\ T\ E' \mid \varepsilon$
- $T \rightarrow F\ T'$
- $T' \rightarrow *\ F\ T' \mid \varepsilon$
- $F \rightarrow (\ E\ ) \mid id$

- FIRST(F) = FIRST(T) = FIRST(E) = {(,id}
- FIRST(T') = {*,ε}
- FIRST(E') = {+,ε}

- FOLLOW(E) =
- FOLLOW(E') =
- FOLLOW(T) =
- FOLLOW(T') =
- FOLLOW(F) =

Example 6

- $E \rightarrow T\ E'$
- $E' \rightarrow +\ T\ E' \mid \varepsilon$
- $T \rightarrow F\ T'$
- $T' \rightarrow *\ F\ T' \mid \varepsilon$
- $F \rightarrow (\ E\ ) \mid id$

- FOLLOW(E) = {$,)}
- FOLLOW(E') = {$,)}
- FOLLOW(T) = {+,$,)}
- FOLLOW(T') = {+,$,)}
- FOLLOW(F) = {*,+,$,)}

- FIRST(F) = FIRST(T) = FIRST(E) = {(,id}
- FIRST(T') = {*,ε}
- FIRST(E') = {+,ε}

Example 7

- $S \rightarrow A\ B\ C \mid A\ D$
- $A \rightarrow a \mid a\ A$
- $B \rightarrow b \mid c \mid \varepsilon$
- $C \rightarrow D\ a\ C$
- $D \rightarrow b\ b \mid c\ c$

- FOLLOW(S) =
- FOLLOW(A) =
- FOLLOW(B) =
- FOLLOW(C) =
- FOLLOW(D) =

- FIRST(D) = FIRST(C) = {b,c}
- FIRST(B) = {b,c,ε}
- FIRST(A) = FIRST(S) = {a}

Example 7

- $S \rightarrow A\ B\ C \mid A\ D$
- $A \rightarrow a \mid a\ A$
- $B \rightarrow b \mid c \mid \varepsilon$
- $C \rightarrow D\ a\ C$
- $D \rightarrow b\ b \mid c\ c$

- FOLLOW(S) = {\$}
- FOLLOW(A) = {b,c}
- FOLLOW(B) = {b,c}
- FOLLOW(C) = {\$}
- FOLLOW(D) = {a,\$}

- FIRST(D) = FIRST(C) = {b,c}
- FIRST(B) = {b,c,ε}
- FIRST(A) = FIRST(S) = {a}

Writing an LL(1) Grammar

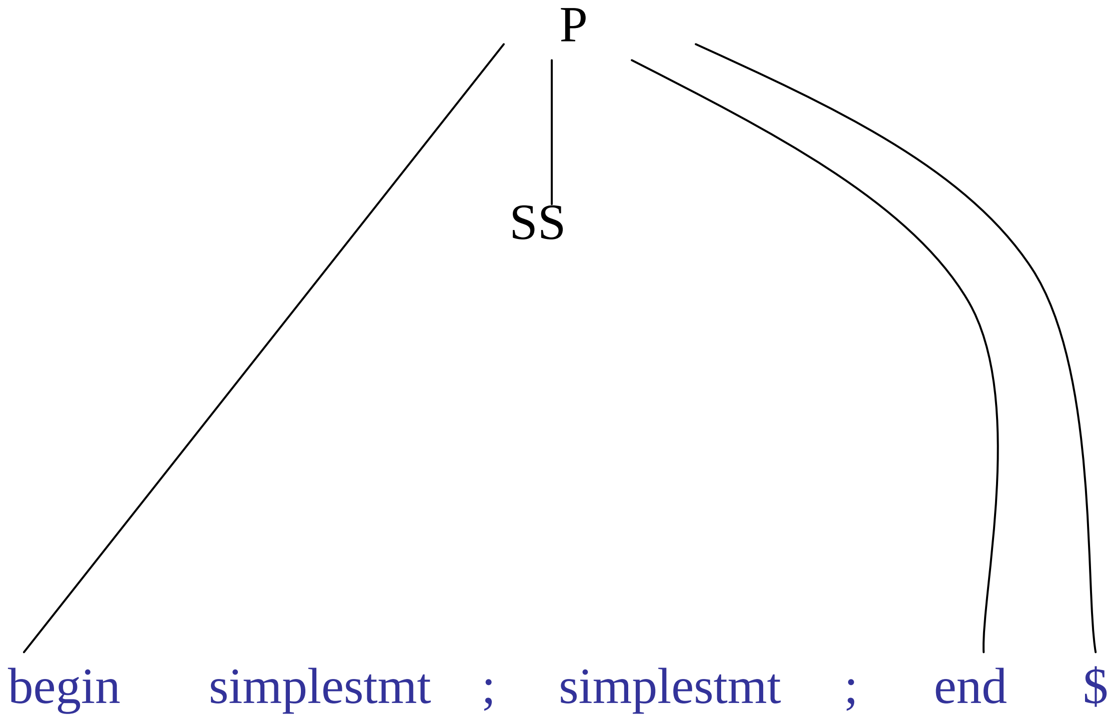
- The two most common obstacles to "LL(1)-ness" are
  - Left recursion
  - Common prefixes

Top Down (LL) Parsing

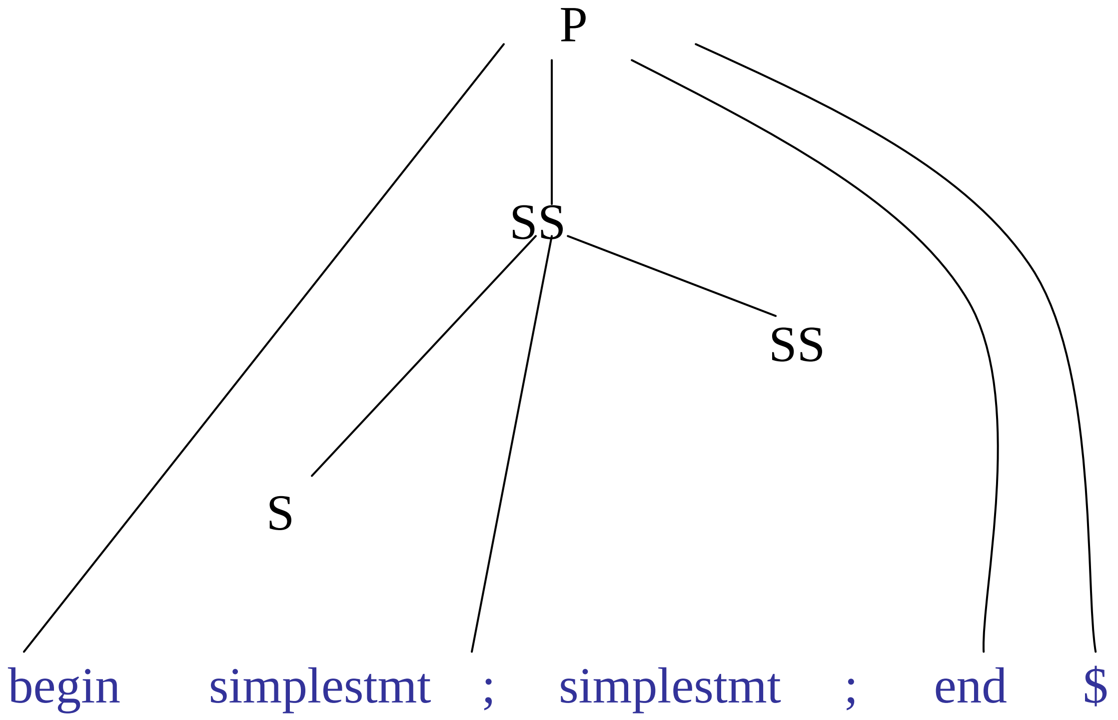
P

begin simplestmt ; simplestmt ; end $

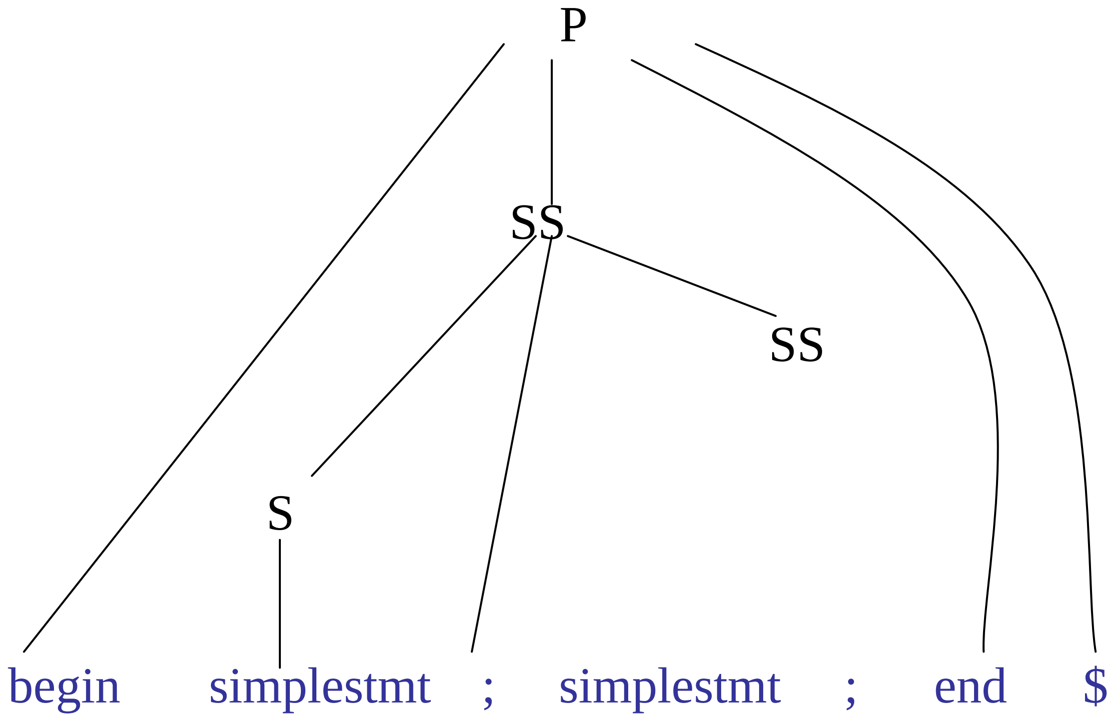
Top Down (LL) Parsing



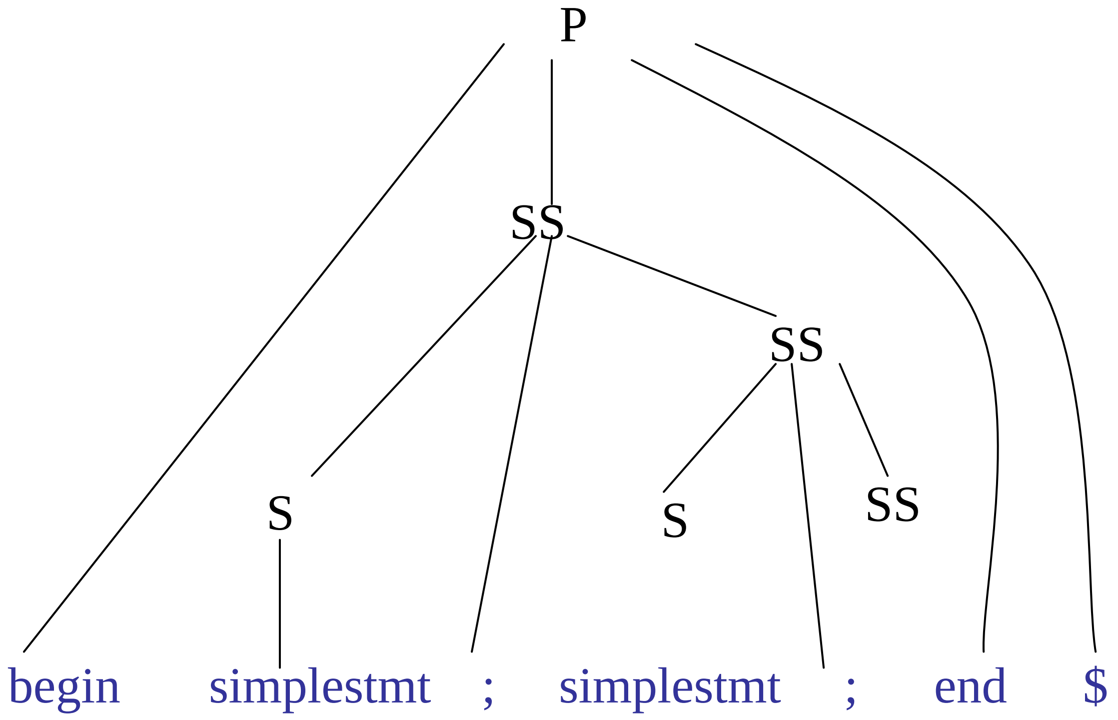
Top Down (LL) Parsing



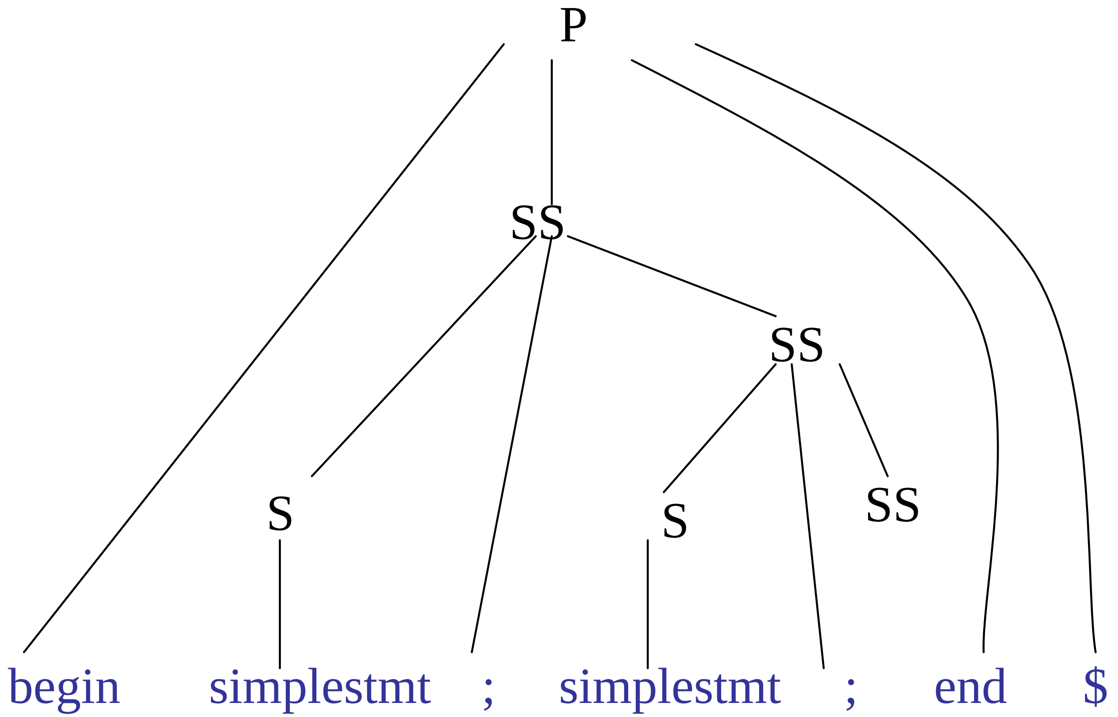
Top Down (LL) Parsing



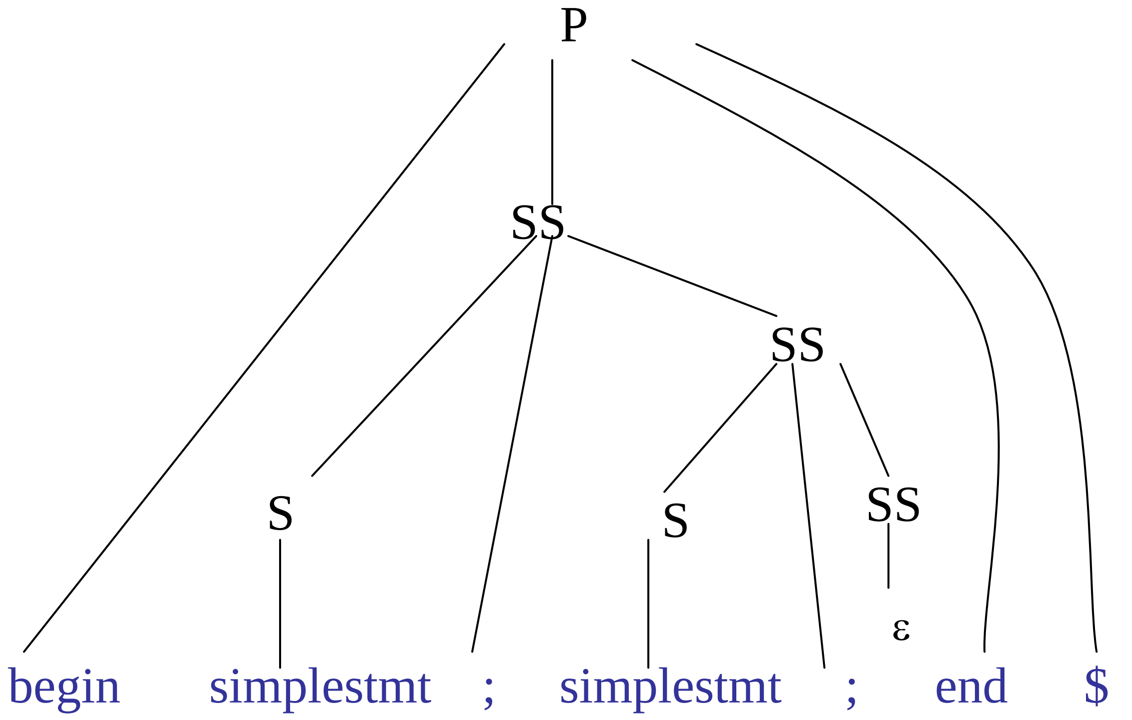
Top Down (LL) Parsing



Top Down (LL) Parsing



Top Down (LL) Parsing



Grammar

$$
\begin{aligned}
S &\rightarrow a\ B \\
&\quad|\ \ b\ C \\
B &\rightarrow b\ b\ C \\
C &\rightarrow c\ c
\end{aligned}
$$

Two strings in the language: abbcc and bcc

Can choose between them based on the first character of the input.

LL(*k*) parsing

- Process input *k* symbols at a time.
- Initially, current non-terminal is start symbol.
- Algorithm
  - Given next k input tokens and current non-terminal T, choose a rule R (T → …)
  - For each element X in rule R from left to right,

    if X is a non-terminal, call function for X

    else if symbol X is a terminal, see if next input symbol matches X; if so, update from the input
- Typically, we consider LL(1)

Two Approaches

- Recursive Descent parsing
  – Code tailored to the grammar
- Table Driven – predictive parsing
  – Table tailored to the grammar
  – General Algorithm

Writing a Recursive Descent Parser

- Procedure for each non-terminal.

  Use next token (lookahead) to choose which production to mimic.
  - for non-terminal X, call procedure X()
  - for terminals X, call 'match(X)'
- `match(symbol) {`

```
  if (symbol = lookahead)
      lookahead = yylex()
  else error()  }
```

- Call `yylex()` before the first call to get first lookahead.

Back to grammar

```
S() {
    if (lookahead==a) { match(a);B(); }
   else if (lookahead == b) { match(b);
     C(); }
   else error("expecting a or b");
}
B() {match(b); match(b); C();}
C() { match(c) ; match(c) ;}

main() {
    lookahead==yylex();
   S();
}
```

S → a B
 | b C

B → b b C

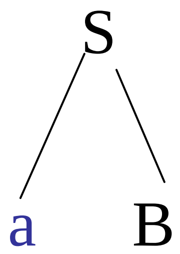
C → c c

Parsing abbcc

S

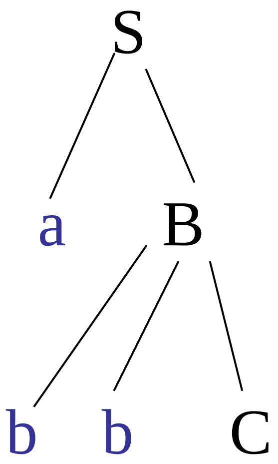
Remaining input: abbcc

Parsing abbcc



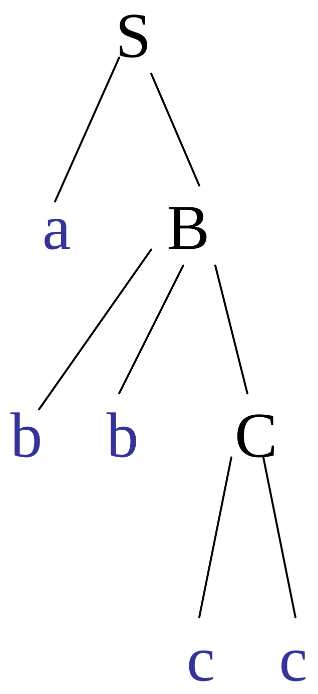
Remaining input: bbcc

Parsing abbcc



Remaining input: cc

Parsing abbcc



Remaining input:

How do we find the lookaheads?

- Can compute PREDICT sets from FIRST and FOLLOW
- PREDICT(A → α) =

  FIRST(α) – {ε} U FOLLOW(A) if ε in FIRST(α)

  FIRST(α) if ε not in FIRST(α)

NOTE: ε never in PREDICT sets

For LL(*k*) grammars, the PREDICT sets for a given non-terminal will be disjoint.

Example

Production	Predict
E → T E’	= FIRST(T) = {(,id}
E’ → + T E’	{+}
E’ → ε	= FOLLOW(E’) = {$,)}
T → F T’	= FIRST(F) = {(,id}
T’ → * F T’	{*}
T’ → ε	= FOLLOW(T’) = {+,$,)}
F → id	{id}
F → (E)	{(}

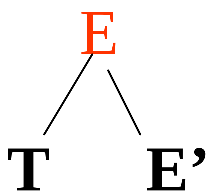
- FIRST(F) = {(,id}
- FIRST(T) = {(,id}
- FIRST(E) = {(,id}
- FIRST(T’) = {*,ε}
- FIRST(E’) = {+,ε}
- FOLLOW(E) = {$,)}
- FOLLOW(E’) = {$,)}
- FOLLOW(T) = {+$,)}
- FOLLOW(T’) = {+,$,)}
- FOLLOW(F) = {*,+,$,)}

Parsing a + b * c

E

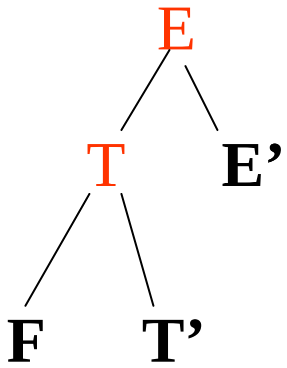
Remaining input: a+b*c

Parsing a + b * c



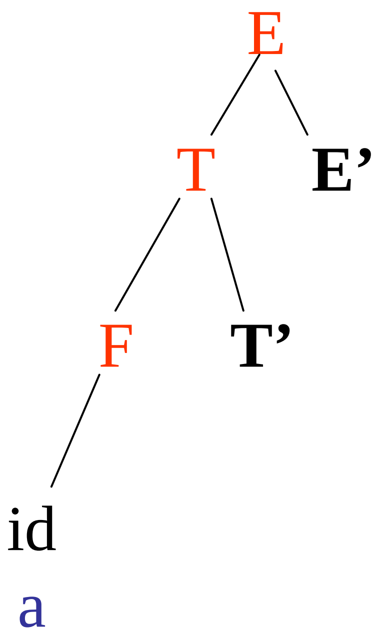
Remaining input: a+b*c

Parsing a + b * c



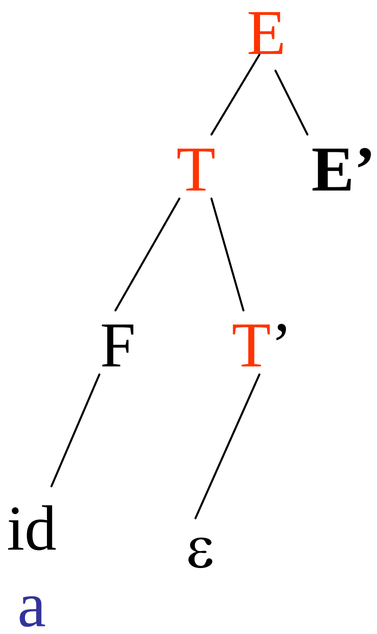
Remaining input: a+b*c

Parsing a + b * c



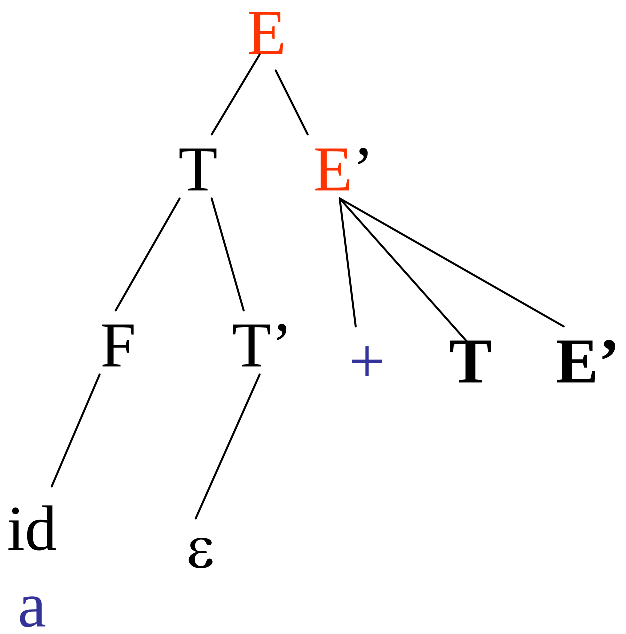
Remaining input: +b*c

Parsing a + b * c



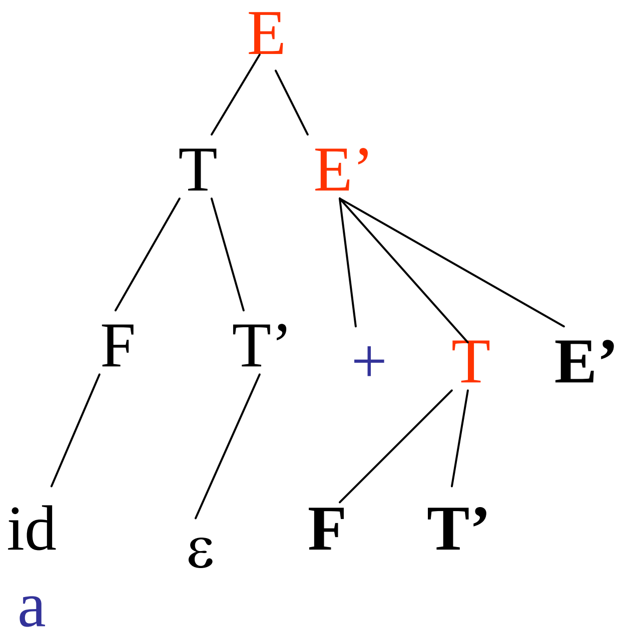
Remaining input: +b*c

Parsing a + b * c



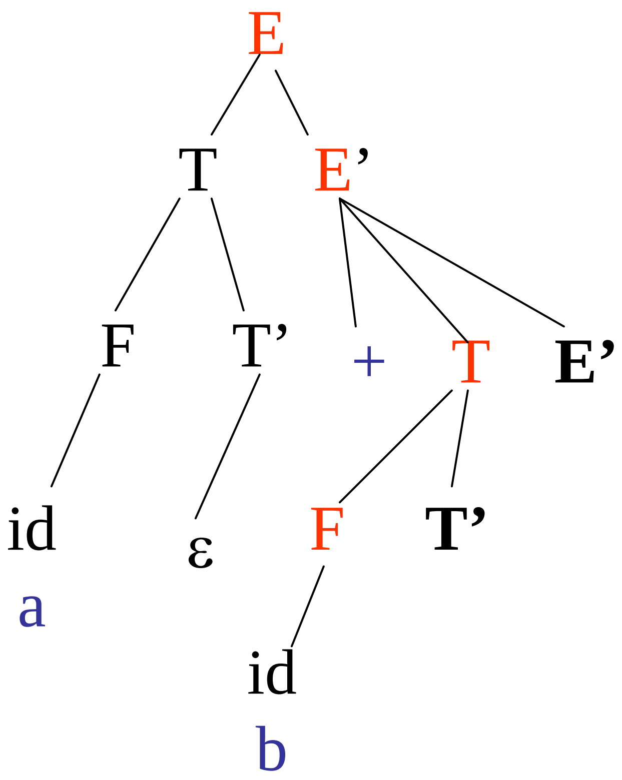
Remaining input: b*c

Parsing a + b * c



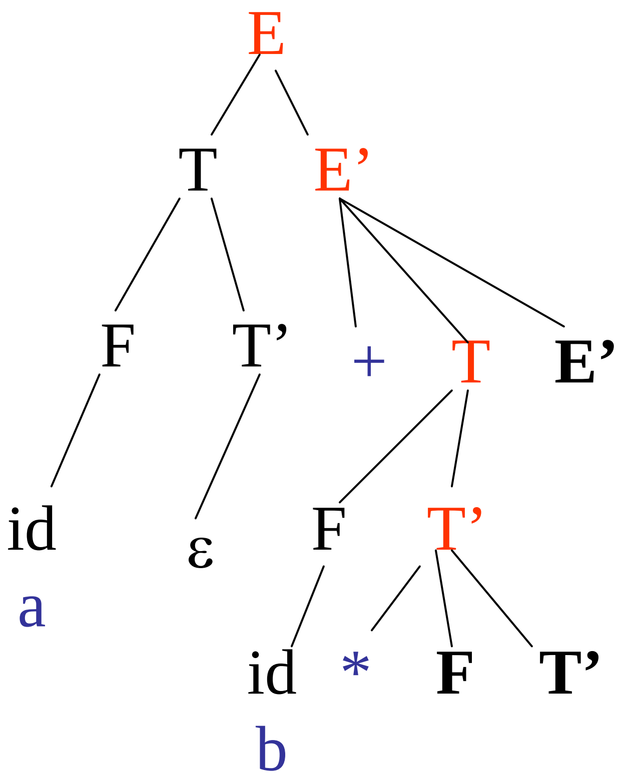
Remaining input: b*c

Parsing a + b * c



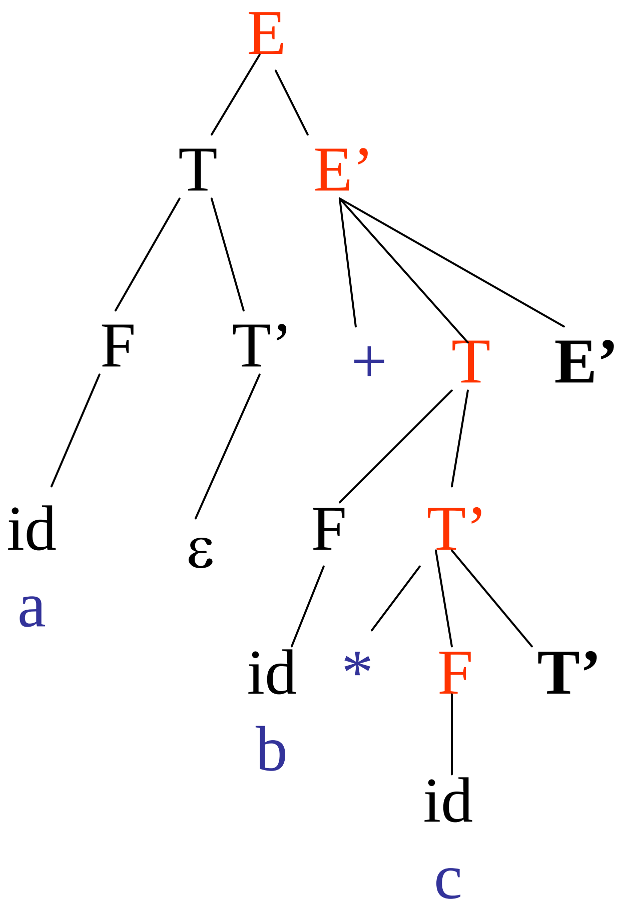
Remaining input: *c

Parsing a + b * c



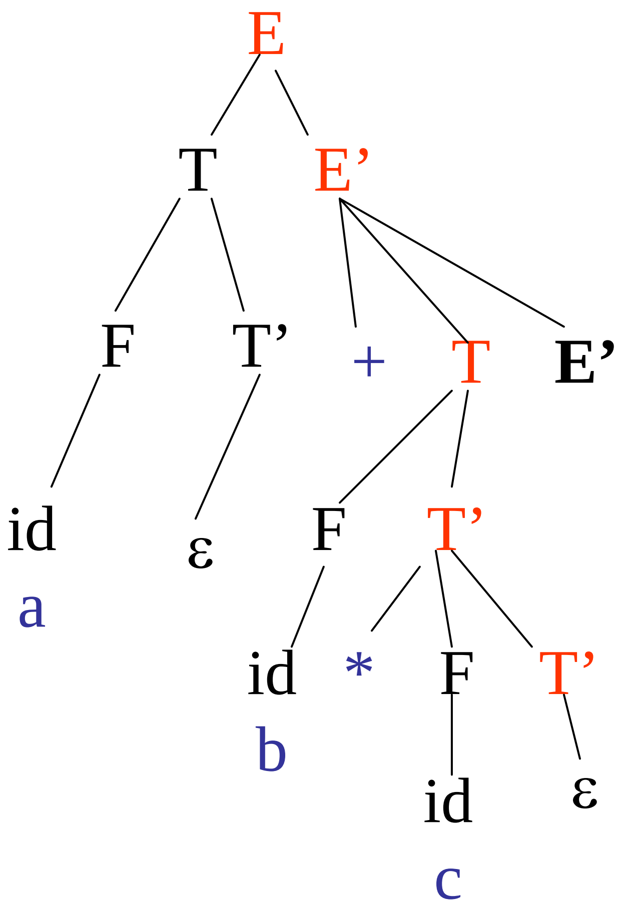
Remaining input: c

Parsing a + b * c



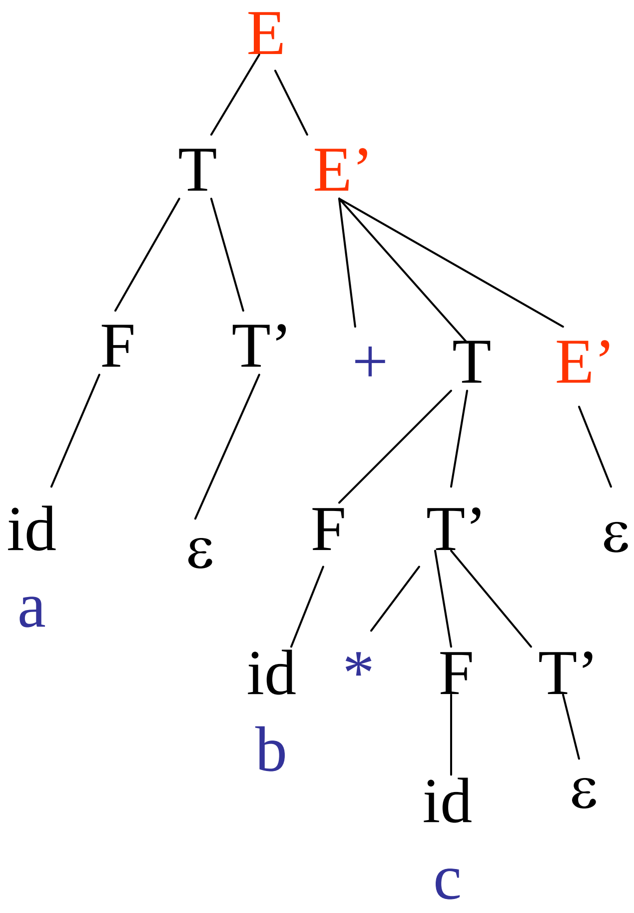
Remaining input:

Parsing a + b * c



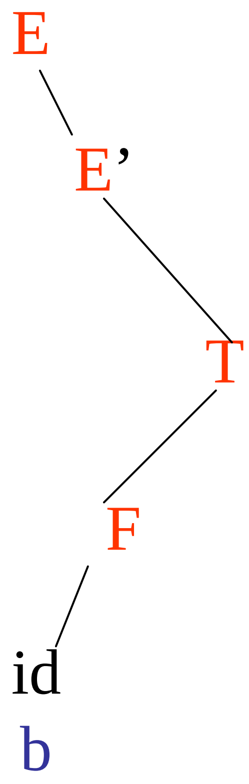
Remaining input:

Parsing a + b * c



Remaining input:

Stacks in Recursive Descent Parsing



- Runtime stack
- Procedure activations correspond to a path in parse tree from root to some interior node

LL(1) Predictive Parse Tables

An LL(1) Parse table is a mapping T: V_n x V_t → production P or error

1. For all productions A → α do
   - For each terminal a in Predict(A →α),
     T[A][a] = A → α
2. Every undefined table entry is an error.

Using LL(1) Parse Tables

ALGORITHM

INPUT: token sequence to be parsed, followed by ‘$’ (end of file)

DATA STRUCTURES:

- Parse stack: Initialized by pushing ‘$’ and then pushing the start symbol
- Parse table T

```
push($); push(start_symbol); lookahead = yylex()
repeat
  X = pop(stack)
  if X is a terminal symbol or $ then
    if X = lookahead then
       lookahead = yylex()
    else error()
  else  /* X is non-terminal */
    if T[X][lookahead] = X → Y1 Y2 ...Ym
        push(Ym) ... push (Y1)
    else error()
 until X = $ token
```

Expression Grammar

NT/T	+	*	(	)	ID	$
E			→ T E'		→ T E'	
E'	→ + T E'			→ ε		→ ε
T			→ F T'		→ F T'	
T'	→ ε	→ * F T'		→ ε		→ ε
F			→ (E)		→ ID	

Parsing a + b * c

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’T+	+b*c$	match
$E’T	b*c$	T → F T’

Stack	Input	Action
$E’T’F	b*c$	F → id
$E’T’id	b*c$	match
$E’T’	*c$	T’ → * F T’
$E’T’F*	*c$	match
$E’T’F	c$	F → id
$E’T’id	c$	match
$E’T’	$	T’ → ε
$E’	$	E’ → ε
$	$	accept

Stack in Predictive Parsing

- Algorithm data structure
- Holds terminals and non-terminals from the grammar
  - terminals – still need to be matched from the input
  - non-terminals – still need to be expanded

Making a grammar LL(1)

- Not all context free languages have LL(1) grammars
- Can show a grammar is not LL(1) by looking at the predict sets
  - For LL(a) grammars, the PREDICT sets for a given non-terminal will be disjoint.

Example

Production	Predict
E → E + T	= FIRST(E) = {(,id}
E → T	= FIRST(T) = {(,id}
T → T * F	= FIRST(T) = {(,id}
T → F	= FIRST(F) = {(,id}
F → id	= {id}
F → (E)	= {(}

- FIRST(F) = {(,id}
- FIRST(T) = {(,id}
- FIRST(E) = {(,id}
- FIRST(T) = {*,ε}
- FIRST(E') = {+,ε}
- FOLLOW(E) = {$,)}
- FOLLOW(E') = {$,)}
- FOLLOW(T) = {+$,)}
- FOLLOW(T') = {+,$,)}
- FOLLOW(F) = {*,+,$,)}

Two problems: E and T

Making a non-LL(1) grammar LL(1)

- Eliminate common prefixes

  Ex: A → B a C D | B a C E

- Transform left recursion to right recursion

  Ex: E → E + T | T

Eliminate Common Prefixes

- $A \rightarrow \alpha\ \beta \mid \alpha\ \delta$

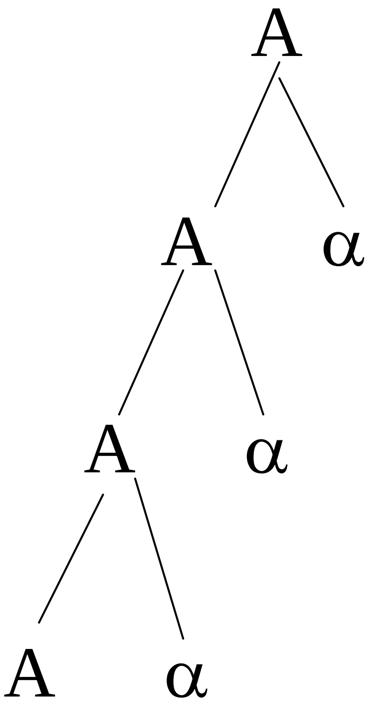
Can become:

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta \mid \delta$

Doesn't always remove the problem. *Why?*

Why is left recursion a problem?



Remove Left Recursion

$A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ \ldots \mid \beta_1 \mid \beta_2 \mid \ldots$

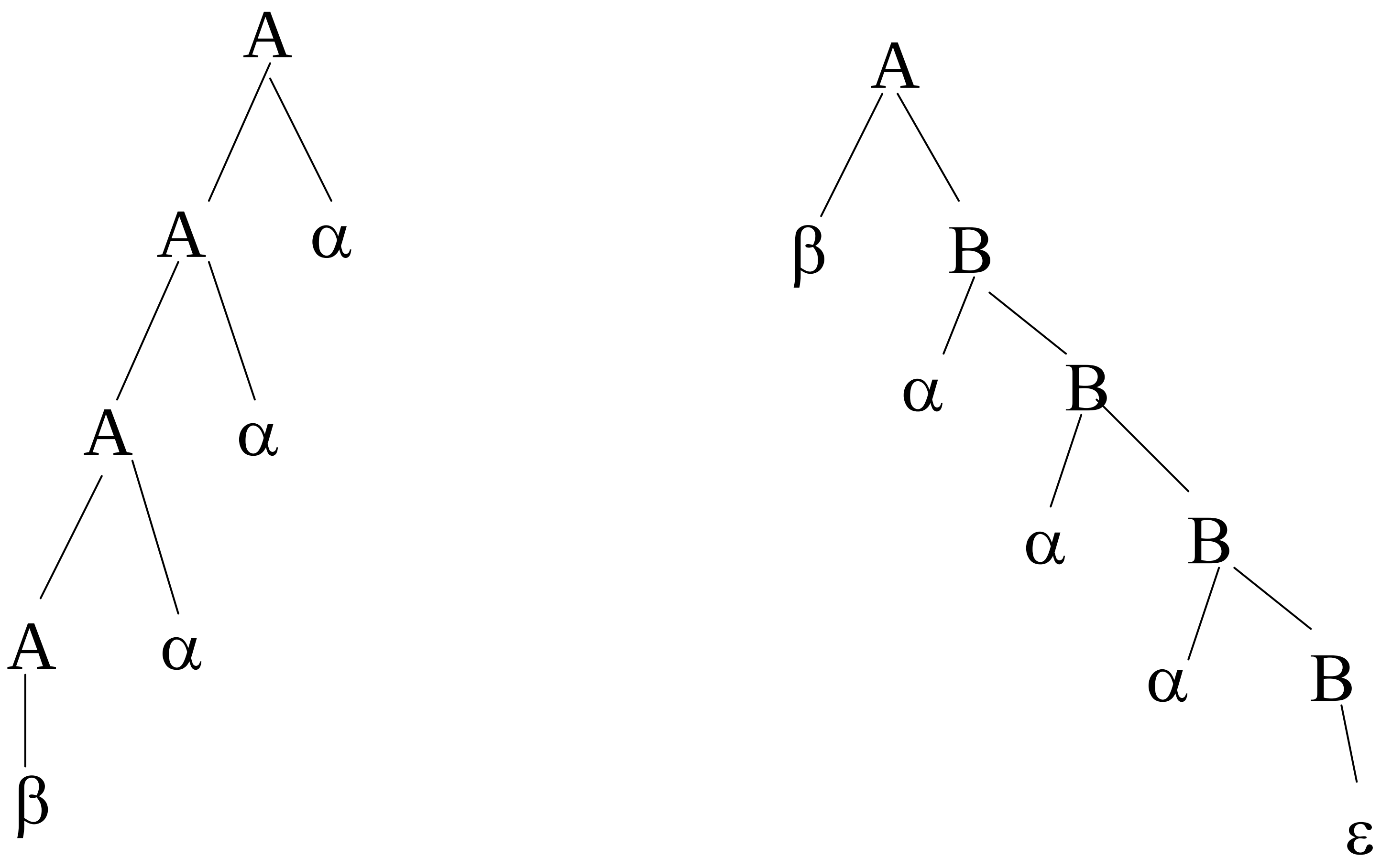
becomes

$A \rightarrow \beta_1\ A' \mid \beta_2\ A' \mid \ldots$

$A' \rightarrow \alpha_1\ A' \mid \alpha_2\ A' \mid \ldots \mid \varepsilon$

The left recursion becomes right recursion

$A \rightarrow A\ \alpha \mid \beta$ becomes $A \rightarrow \beta\ B$, $B \rightarrow \alpha\ B \mid \varepsilon$



Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Bottom-up Parsing (cont.)

- Intuition about handles:
  - Def: β is the *handle* of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$
  - Def: β is a *phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
  - Def: β is a *simple phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

Bottom-up Parsing (cont.)

- Intuition about handles:
  - The handle of a right sentential form is its leftmost simple phrase
  - Given a parse tree, it is now easy to find the handle
  - Parsing can be thought of as handle pruning

Bottom-up Parsing (cont.)

- Shift-Reduce Algorithms
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing (cont.)

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  - They can detect syntax errors as soon as it is possible.
  - The LR class of grammars is a superset of the class parsable by LL parsers.

Bottom-up Parsing (cont.)

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  - There were only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Bottom-up Parsing (cont.)

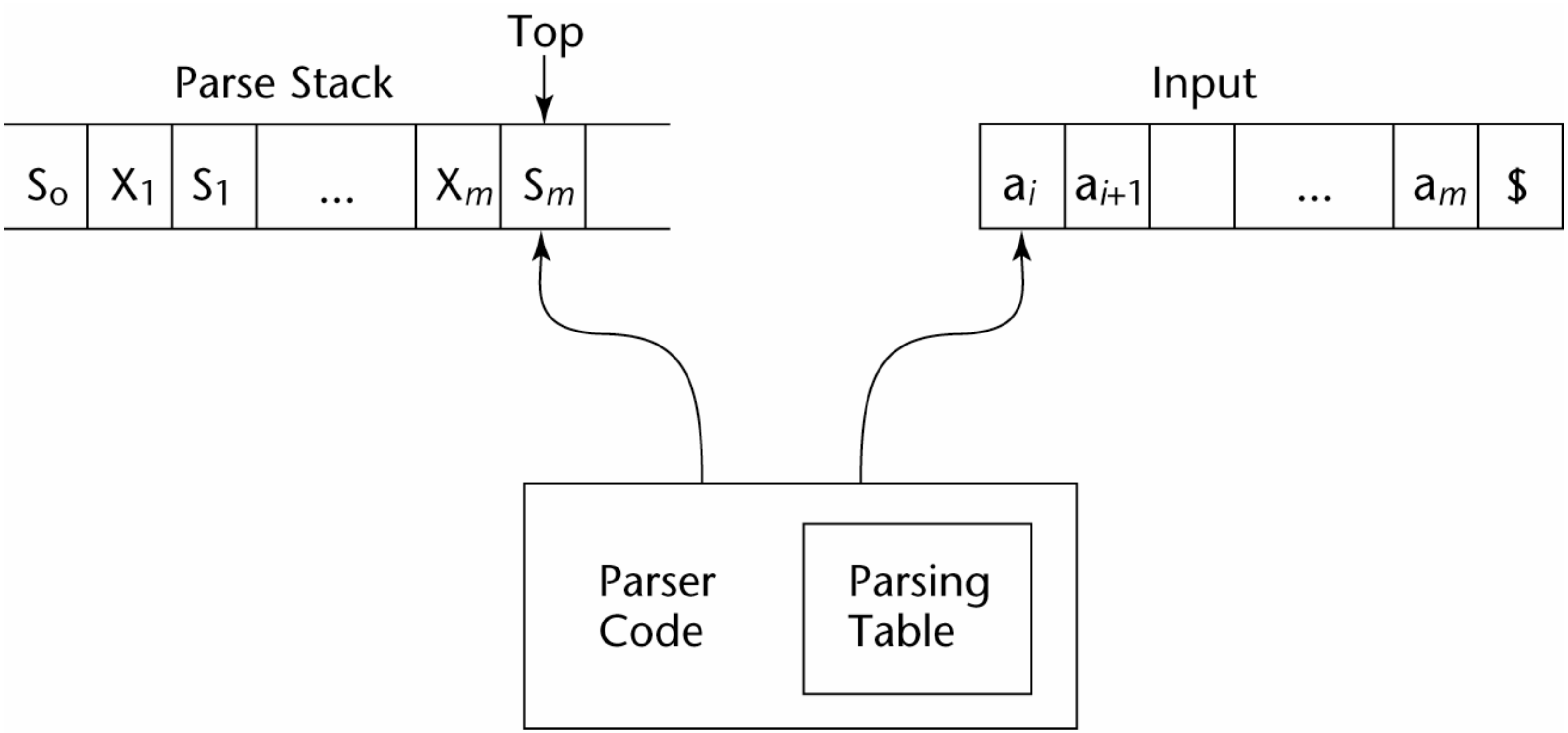
- An LR configuration stores the state of an LR parser

$(S_0 X_1 S_1 X_2 S_2 \ldots X_m S_m,\ a_i a_i{+}1 \ldots a_n \$)$

Bottom-up Parsing (cont.)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals

Structure of An LR Parser



Bottom-up Parsing (cont.)

- Initial configuration: $(S_0, a_1 \ldots a_n\$)$
- Parser actions:
  - If ACTION$[S_m, a_i]$ = Shift S, the next configuration is:

    $(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS, a_{i+1} \ldots a_n\$)$
  - If ACTION$[S_m, a_i]$ = Reduce $A \to \beta$ and S = GOTO$[S_{m-r}, A]$, where r = the length of β, the next configuration is

    $(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$

Bottom-up Parsing (cont.)

- Parser actions (continued):
  - If ACTION[S_m, a_i] = Accept, the parse is complete and no errors were found.
  - If ACTION[S_m, a_i] = Error, the parser calls an error-handling routine.

LR Parsing Table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing (cont.)

- A parser table can be generated from a given grammar with a tool, e.g., **yacc**

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
  - Detects syntax errors
  - Produces a parse tree
- A recursive-descent parser is an LL parser
  - EBNF
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach