Chapter 4

Lexical and Syntax Analysis

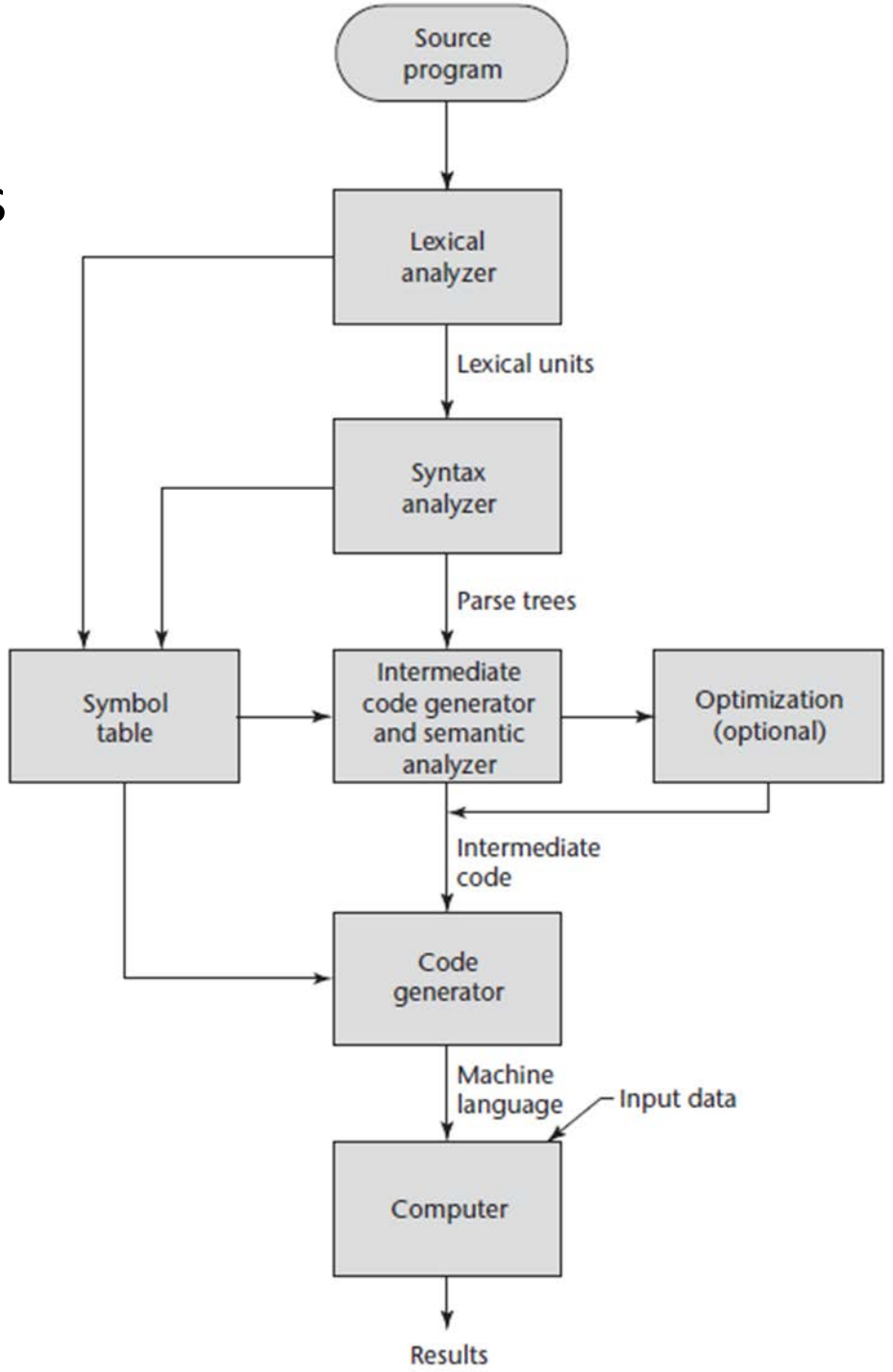
Topics

- 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)

Recall:
The Compilation Process



Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - *Lexical analyzer*, or *Scanner*: a low-level part
    - mathematically, a *Finite Automaton* based on a *Regular Grammar* – CSci 435.
  - *Syntax analyzer*, or *Parser*: a high-level part
    - mathematically, a *Push-Down Finite Automaton* based on a *Context-Free Grammar* in BNF – CSci 435.

Syntax Analysis: The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors;

    for each error,

    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*
- Useful parsers look only *one token ahead* in the input

The Parsing Problem (cont.)

- Notational convention for grammar symbols/strings
  - Let Σ be a set of *alphabet*.
  - *Terminal symbols*:
    - lowercase letter at the beginning of Σ, e.g.) (*a, b, ….)*
  - *Nonterminal symbols (i.e. Variables)*:
    - uppercase letter at the beginning of Σ, e.g.) (A, B, ….)
  - *Terminals or Nonterminals*:
    - uppercase letter at the end of Σ, e.g.) (W, X, Y, Z)
  - *Strings of terminals*:
    - lowercase letters at the end of the alphabet, e.g.) (w, x, y, z)
  - *Mixed strings* (terminals and/or nonterminals):
    - lowercase Greek letters, e.g.) (α, β, γ, δ)
    - used in the sentential form (chap3-syntax-#9)

The Parsing Problem (cont.)

- Top-Down (TD) Parser
  - It traces/builds a parse tree in preorder; leftmost derivation.
  - Given a sentential form, $xA\alpha$, (x is a string of terminals; α is mixed)

    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.

  e.g.) A-rules: $A \rightarrow bB \mid cBb \mid a$.

  Then, $xA\alpha \Rightarrow xbB\alpha \mid xcBb\alpha \mid xa\alpha$
  - Parsing decision problem for TD-parser.
- Common TD-parsing algorithms: *LL algorithms*
  - *Recursive descent parser* - a coded implementation based on BNF.
  - *LL parsers* (*Left*-to-right, *Leftmost* derivation)
    - Read the symbols left-to-right; derive from the *leftmost* variable.
    - table-driven implementation - use a *parsing table.*

The Parsing Problem (cont.)

- Bottom-Up (BU) parser
  - It constructs a parse tree by beginning at the *leaves* and progressing *toward the root*.
  - The *reverse of rightmost derivation*: the sentential forms of the derivation are produced in order of last to first.
  - Given a right sentential form, α ,
    the parser decides what *substring of* α is the RHS of the rule in the grammar that must be reduced to produce the previous sentential form in the right-most derivation.
    - E.g.) $S \Rightarrow^{*} \gamma \Rightarrow \alpha \Rightarrow \ldots \Rightarrow w$, where $\alpha = \alpha_1 \beta \alpha_2$, $\gamma = \alpha_1 A \alpha_2$, by $A \rightarrow \beta$.
    - A given right sentential form may include multiple RHS
      → Find the correct RHS, *handle*, to reduce.
  - The most common bottom-up parsing algorithms are in the LR family – *LR parser* (*Left*-to-right scan, *Rightmost* derivation)

The Parsing Problem (cont.)

- Bottom-Up parser (cont.)
  - E.g.) S $\rightarrow a$Ac, A $\rightarrow a$A | b

    For a string *aabc*, aabc $\Rightarrow^1$ *a*aAc $\Rightarrow^2$ *a*Ac $\Rightarrow^3$ S
  - 1. Find the handle in *aabc*.
    - Only RHS 'b' in A $\rightarrow$ b $\Rightarrow$ Replace *b* with its LHS.
  - 2. Find the handle in the right sentential form *aaAc*.
    - Examine the symbols on or both sides of a possible handle.
    - RHS '*a*A' in A $\rightarrow$ *aA* $\Rightarrow$ Replace *aA* with its LHS.
    - ? RHS '*a*Ac' in S $\rightarrow$ *aAc* *?* $\Rightarrow$ *a*S -- no handle in step 3 - failure
  - 3. Find the handle in the right sentential form *aAc*.
    - RHS '*a*Ac' in S $\rightarrow$ *aAc* $\Rightarrow$ Replace *aAc* with its LHS.
    - ? RHS '*a*A' in A $\rightarrow$ *aA* *?* $\Rightarrow$ *Ac* -- no handle in step 4 – failure.
  - 4. Find the handle in the right sentential form S.
    - S is the start variable, the root of the parse tree. Done!

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, ($|w| = n$).
    - CYK algorithm – CSci 435.
  - 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.
    - Faster algorithm but less general.

Recursive-Descent Parsing: Top-Down parser

- Recursive-Descendent Top-Down parser.
  - a coded implementation based on BNF.
- A recursive-descent parser consists of a collection of *subprograms* and produces a parse tree in *top-down* order.
  - many subprograms are recursive.
  - 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 | int_constant | ( <expr> )
```

Recursive-Descent (RD) Parsing (cont.)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`.
- RD subprogram for a rule with a *single RHS*:
  - For each *terminal* symbol in the RHS,
    compare it with the `nextToken`;
    if they match, continue – `lex` is called to get next token;
    else there is an error.

    Note: the token codes are defined as named constants.
  - 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() {
  print("Enter <expr> \n");

/* Parse the first term */
  term();

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

  while (nextToken == ADD_OP ||
         nextToken == SUB_OP){
    lex();
    term();
  }
  print("Exit <expr> \n");
}
```

- This routine `expr()` does not detect errors.
- Convention:

  Every parsing routine leaves the next token in `nextToken`, which has the code for the *leftmost token of the input* that has not yet been used in the parsing process.

- `<expr>` : a function that parses a language generated by a nonterminal `<expr>`
- `<term>` : a function that parses a language generated by a nonterminal `<term>` of the grammar.

Recursive-Descent Parsing (cont.)

- RD-subprogram for a nonterminal with *more than one RHS:*
  - An initial process to determine *which RHS* it is to parse.
  - The correct RHS is chosen on the basis of the *nextToken* of input (the lookahead).
  - The *nextToken* 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.
    - For the detected error, it produces a diagnostic message and parser must recover from it so that the parsing process can continue.

Recursive-Descent Parsing (cont.)

```
/* term
Parses strings in the language generated
by the rule:
<term> -> <factor> {(* | /) <factor>)}
*/
void term() {
  print("Enter <term> \n");

/* Parse the first factor */
  factor();

/* As long as the next token is * or /,
   next token and parse the next factor */
  while (nextToken == MULT_OP
         || nextToken == DIV_OP){
    lex();
    factor();
  }

  printf("Exit <term> \n");
} /* End of function term */
```

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

void factor( )   {
    print("Enter <factor> \n");
/*  Determine which RHS  */
      if (nextToken == ID_CODE
       || nextToken == INT_CODR )

          /*   Get the next token   */
            lex( ) ;

       /* If the RHS is (<expr>)
          call lex to pass over the left ),
          call expr, and check the right (  */
       else if (nextToken == LP_CODE) {
                lex( );
                expr( );
                if (nextToken == RP_CODR)
                     lex( );
                else
                     error( );
             }    /* end of else if  */

      else error();  /* Neither RHS matches */
    printf("Exit <factor> \n");
}
```

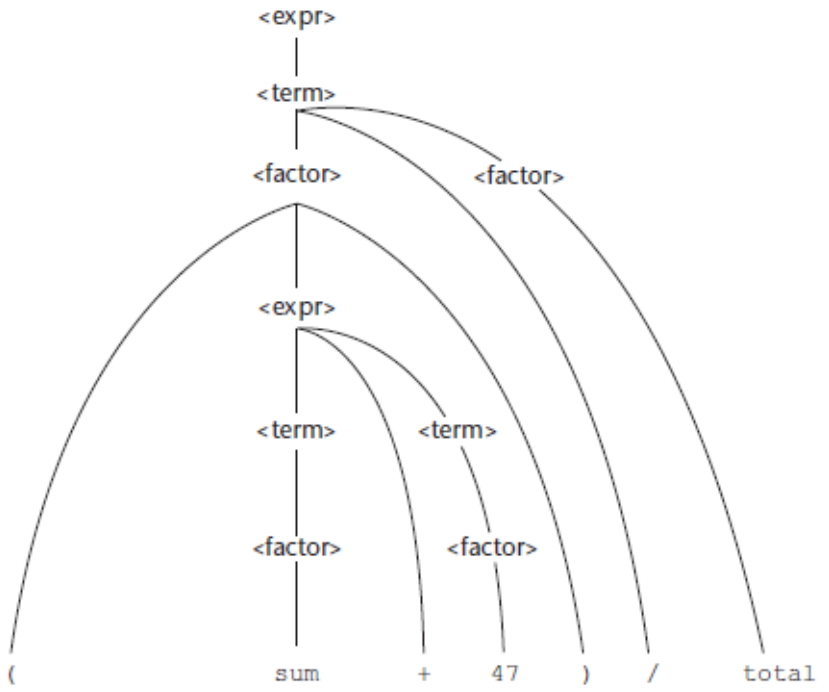
Recursive-Descent Parsing (cont.)

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

Parsing begins by calling `lex` and the start symbol routine, `expr`.

- Trace of the lexical and syntax analyzers on `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```



Recursive-Descent Parsing: Limitation/Restriction

- The LL Grammar Class

  1. *The Left Recursion Problem*

     - If a grammar has direct/indirect *left recursion*,
       it can**not** be the basis for a top-down parser.

       – Modify the grammar to remove direct left recursion:

       For each nonterminal, A,

       1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
          where none of the β's begins with A

       2. Replace the original A-rules with

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

          $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$ - an erasure rule (end case)

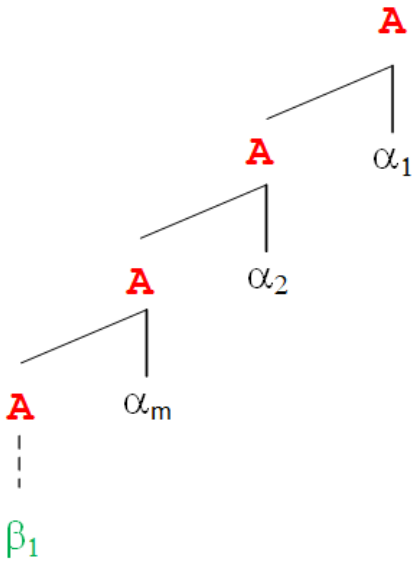
          (a use of ε effectively erases its LHS from the sentential form)

Recursive-Descent Parsing: Limitation/Restriction

- The LL Grammar Class

1. *The Left Recursion Problem*

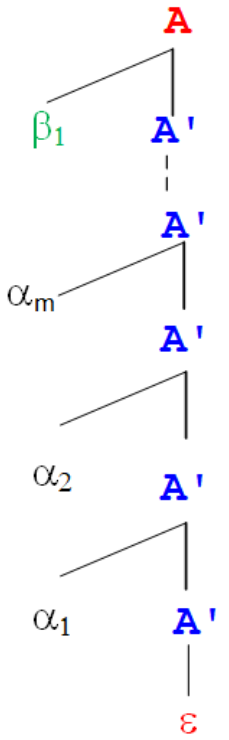
$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$



$\beta_1 \ldots . \alpha_m \alpha_2 \alpha_1$

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

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



$\beta_1 \ldots . \alpha_m \alpha_2 \alpha_1$

Recursive-Descent Parsing: Limitation (cont.)

- Example: Conversion of a *direct* left recursive rule
  - $E \rightarrow E + T \mid T, \quad T \rightarrow T * F \mid F, \quad F \rightarrow (E) + id$
  - For E-rules, $\alpha_1 = + T$ and $\beta = T$: replace with
    - $E \rightarrow T\ E', \quad E' \rightarrow + T\ E' \mid \varepsilon$
  - For T-rules, $\alpha_2 = * F$ and $\beta = F$: replace with
    - $T \rightarrow F\ T', \quad T' \rightarrow * F\ T' \mid \varepsilon$
  - F rule: no left recursion - no change
  - The complete replacement grammar is:
    - $E \rightarrow T\ E'$,
    - $E' \rightarrow + T\ E' \mid \varepsilon$
    - $T \rightarrow F\ T'$,
    - $T' \rightarrow * F\ T' \mid \varepsilon$
    - $F \rightarrow (E) + id$

Recursive-Descent Parsing: Limitation (cont.)

- Example: An *indirect* left recursive rule
  - $A \rightarrow B\ a\ A \mid T, \quad B \rightarrow A\ b$
  - i.e. equivalent to $A \rightarrow A\ b\ a\ A \mid T$
- Left recursion:
  - a problem for *ALL* top-down parsing algorithms, not confined to the recursive-descent top-down parsing.
  - NOT a problem for bottom-up parsing algorithm.

Limitation of Recursive-Descent parsing of LL parser:

1. *Left recursion* disallowed top-down parsing.
2. The *lack of pairwise disjointness* disallows top-down parsing.
   - Can the parser always choose the correct RHS based on the *next token*, using only the 1st token generated by the leftmost nonterminal in the current sentential form? -- the pairwise disjointness test

Recursive-Descent Parsing (cont.)

2. Problem with the *lack of pairwise disjointness*.
   - The inability to determine the *correct RHS* on the basis of *one token of lookahead*.
   - Definition: $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$

     (If $\alpha \Rightarrow^* \varepsilon$, $\varepsilon \in \text{FIRST}(\alpha)$)

- 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

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

  - i.e. the first terminal symbol that can be generated in a derivation for each of them must be *unique* to that RHS.

Recursive-Descent Parsing (cont.)

- Example: A $\rightarrow$ *a*B | bAb | Bb, B $\rightarrow$ cB | d
  - FIRST(A) = {*a*}, {b} and {c}, {d} -- pairwise disjoint.

    In terms of a recursive-descent parser,
  - the code of the subprogram for parsing the nonterminal A can choose which RHS it is dealing with by seeing *only the first terminal* symbol of input (token) that is generated by the nonterminal.
- Example: A $\rightarrow$ *a*B | BAb, B $\rightarrow$ *aA* | b
  - FIRST(A) = {*a*} and {*a*}, {*b*} – not pairwise disjoint.

    In terms of a recursive-descent parser,
  - the subprogram for A *could not determine* which RHS was being parsed by looking at the next symbol of input, because if it were an *a*, it could be either RHS.

Recursive-Descent Parsing (cont.)

- Solution: *Left factoring* to resolve the *non-disjointness*.

  Replace (non-pairwise disjoint rule)

  <variable> → *identifier* | *identifier* [<expression>]

  with (expression in brackets (a subscript))

  <variable> → identifier <new>

  <new> → ε | [<expression>]

  or

  <variable> → identifier [[<expression>]]

  (the outer brackets [ ] are metasymbols that indicate what is inside is optional in EBNF)

Bottom-Up Parsing (BU)

- Problem: Find the *correct RHS* (i.e. *handle*)
  in a right-sentential form to reduce
  to the previous right-sentential form in the derivation.
- *LR parser* (*Left*-to-right, *Rightmost* derivation)
- Example: $E \rightarrow E + T \mid T, \quad T \rightarrow T * F \mid F, \quad F \rightarrow (E) \mid id$
  - A left recursive grammar which is acceptable for BU-parser.

  Rightmost derivation:

$$E \Rightarrow \underline{E + T} \Rightarrow E + \underline{T * F} \Rightarrow E + T * \underline{id} \Rightarrow E + \underline{F} * id \Rightarrow \underline{E} + \underline{id} * id$$
$$\Rightarrow \underline{T} + id * id \Rightarrow \underline{F} + id * id \Rightarrow \underline{id} + id * id$$

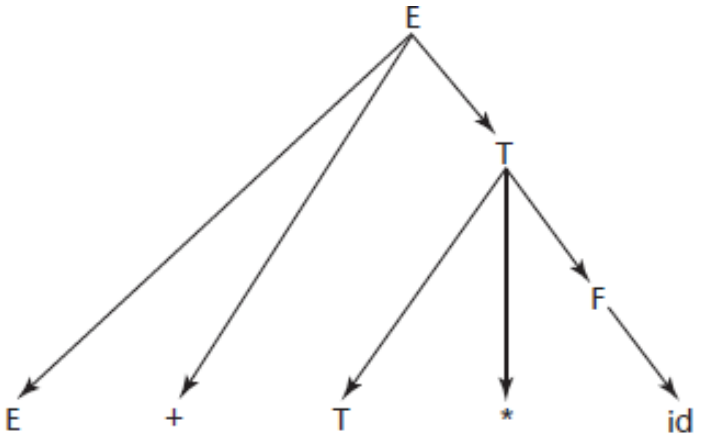
- The RHS is rewritten as its corresponding LHS to get the previous sentential form.
- The process of bottom-up parsing produces the reverse of a rightmost derivation.

Bottom-up Parsing (cont.)

- Intuition about handles: (refer to #8 for the notational convention)
  - 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 $\gamma = \alpha_1\beta\alpha_2$
    if and only if $S \Rightarrow^* \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$
    - A phrase can be derived from a single nonterminal in *one or more* tree levels
  - Def: β is a *simple phrase* of the right sentential form $\gamma = \alpha_1\beta\alpha_2$
    if and only if $S \Rightarrow^* \alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$
    - A phrase that takes a *single derivation step* from its root nonterminal node; it can be derived in just a single tree level.
    - A simple phrase is always a RHS in the grammar.

Bottom-up Parsing (cont.)

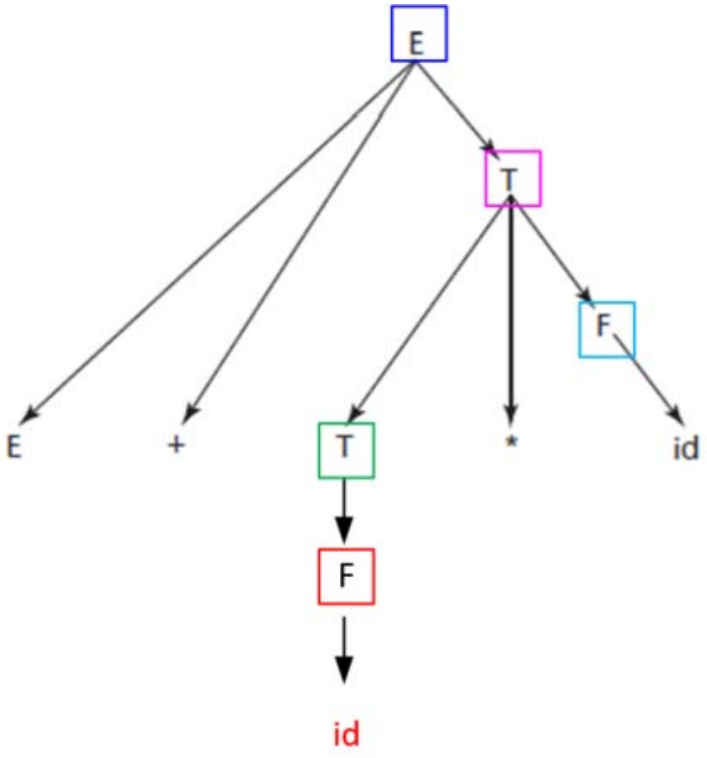
- Intuition about handles:
  - The *handle* of a right sentential form is
    its *leftmost simple phrase.*
  - Given a parse tree, it is easy to find the handle.
  - Parsing can be thought of as handle pruning.



- Sentential form in the leaves:

  `E+T*id`
  - 3 internal nodes → 3 phrases.
  - E generates a phrase `E+T*id`.
    - where $\alpha_1 = \alpha_2 = \varepsilon$
  - T generates a phrase `T*id`.
    - where α_1 = `E+`, $\alpha_2 = \varepsilon$
  - F generates a (simple) phrase `id`.
    - where α_1 = `E+T*`, $\alpha_2 = \varepsilon$
  - So, the phrases of the sentential form are `E+T*id`, `T*id`, `id`.
    - Not necessarily RHS of the underlying grammar.

Bottom-up Parsing (cont.)



- Sentential form in the leaves:

  `E+id*id`

  - 5 internal nodes → 5 phrases.
  - E generates a phrase `E+id*id`.
    - where $\alpha_1 = \alpha_2 = \varepsilon$
  - T generates a phrase `id*id`.
    - where α_1 = `E+`, $\alpha_2 = \varepsilon$
  - T generates a phrase `id`. (in 2 steps)
    - where α_1 = `E+`, α_2 = `*id`.
  - F generates a *simple* phrase `id`.
    - where α_1 = `E+`, α_2 = `*id`
  - F generates a *simple* phrase `id`.
    - where α_1 = `E+id*`, $\alpha_2 = \varepsilon$
  - So, the handle is id, the leftmost simple phrase.

Bottom-up Parsing: LR-Parser (cont.)

- Shift-Reduce Algorithms
  - Bottom-Up parsing algorithm with a stack.
  - Input: the stream of *tokens* of a program;
  - Output: a sequence of *grammar rules*.
  - Reduce: an action of *replacing an RHS* (the *handle*) on the top of the parse stack with its *corresponding LHS*.
  - Shift: an action of *moving the next input token* to the top of the parse *stack*.
  - Every parser for PL is a PushDown Automaton (PDA)
    - a recognizer of Context-Free Language (CFL). – CSci 435.
- LR parsers:
  - A base of most BU-parsing algorithms.
  - Left-to-right and Rightmost derivation in reverse.

Bottom-up Parsing: LR-Parser (cont.)

- Advantages of LR parsers:
  - They will work for *nearly all grammars* that describe prog. 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: LR parser grammar $\supset$ LL parser grammar
    - E.g.) many left-recursive grammars are LR, but not LL.
- Disadvantages of LR parsers:
  - LR parsers are difficult to produce the parsing table by hand; they must be constructed with a tool.

Bottom-up Parsing: LR-Parser (cont.)

- 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 are 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: LR-Parser (cont.)

- An LR configuration stores the state of an LR parser

$$(S_0X_1S_1X_2S_2\ldots X_mS_m,\ a_ia_{i+1}\ldots a_n\$)$$

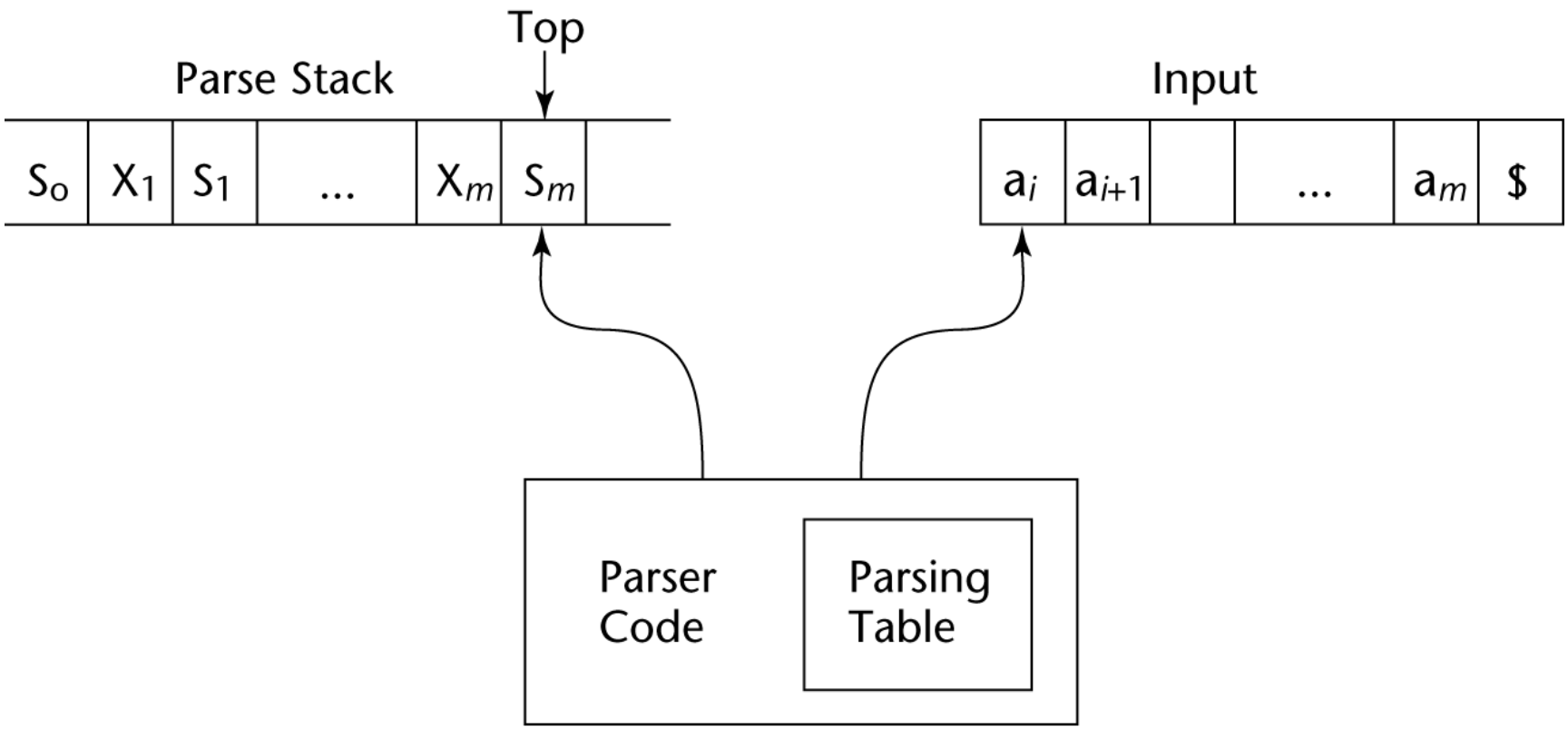
- LR parsers are table-driven, where the table has two components:

  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



$(S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)$

Bottom-up Parsing: LR Parser (cont.)

- Initial configuration: (S_0, $a_1 \ldots a_n$\$)
- Parser actions:

For a current parser state = [state−symbol on the stack top , next input token symbol]

  - Shift (i.e. Push):
    - [next input-symbol, state-symbol] is pushed onto the stack.
      - the state symbol is part of the Shift specification in the Action table.
  - Reduce (RHS to LHS):
    - remove the handle from the stack, along with its state symbols.
    - Push the LHS of the rule.
    - Push the state symbol from the GOTO table, using the state symbol just below the new LHS in the stack and the LHS of the new rule as the row and column into the GOTO table.
  - For an Accept, the parse is complete and no errors were found.
  - For an Error, the parser calls an error-handling routine.

LR Parsing Table

state symbol

	Action : terminal symbol						Goto: Nonterminal symbol		
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			

A parser table can be generated from a given grammar with a tool, e.g., `yacc` or `bison`

LR Parsing: Example

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

	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			

Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

Simplified Example: no state-symbol in Stack (cont.)

- LR(1) Grammar

  $E \rightarrow E + T \mid T$

  $T \rightarrow T \times F \mid F$

  $F \rightarrow (E) \mid a$.

- Rightmost Derivation:

  $E \Rightarrow E + T$

  $\Rightarrow E + F$

  $\Rightarrow E + a$

  $\Rightarrow T + a$

  $\Rightarrow T \times F + a$

  $\Rightarrow T \times a + a$

  $\Rightarrow F \times a + a$

  $\Rightarrow a \times a + a$

- The actions of an LR(*1*) parser:

stack	***input***	***action***
	$a \times a + a\$$	shift
a	$\times a + a\$$	reduce by $F \rightarrow a$
F	$\times a + a\$$	reduce by $T \rightarrow F$
T	$\times a + a\$$	shift
$T \times$	$a + a\$$	shift
$T \times a$	$+ a\$$	reduce by $F \rightarrow a$
$T \times F$	$+ a\$$	reduce by $T \rightarrow T \times F$
T	$+ a\$$	reduce by $E \rightarrow T$
E	$+ a\$$	shift
E +	$a\$$	shift
$E + a$	$\$$	reduce by $F \rightarrow a$
E + F	$\$$	reduce by $T \rightarrow F$
E + T	$\$$	reduce by $E \rightarrow E + T$
E	$\$$	accept

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 the current sentential form.
- The *LR family of shift-reduce parsers* is the most common bottom-up parsing approach.