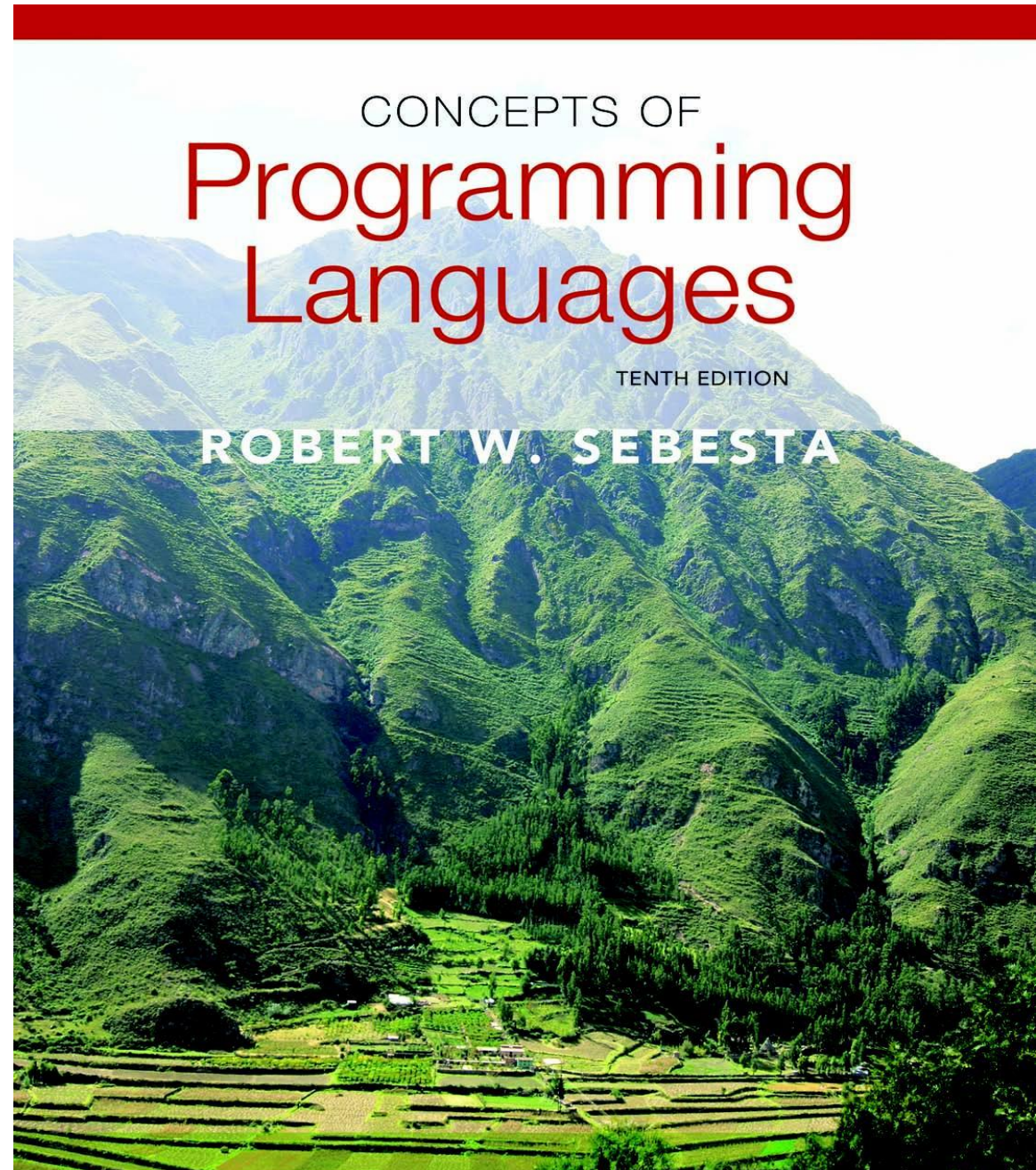


Chapter 4

Lexical and Syntax Analysis





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)

Introduction

- 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)

Introduction

- Reasons to use 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

Introduction

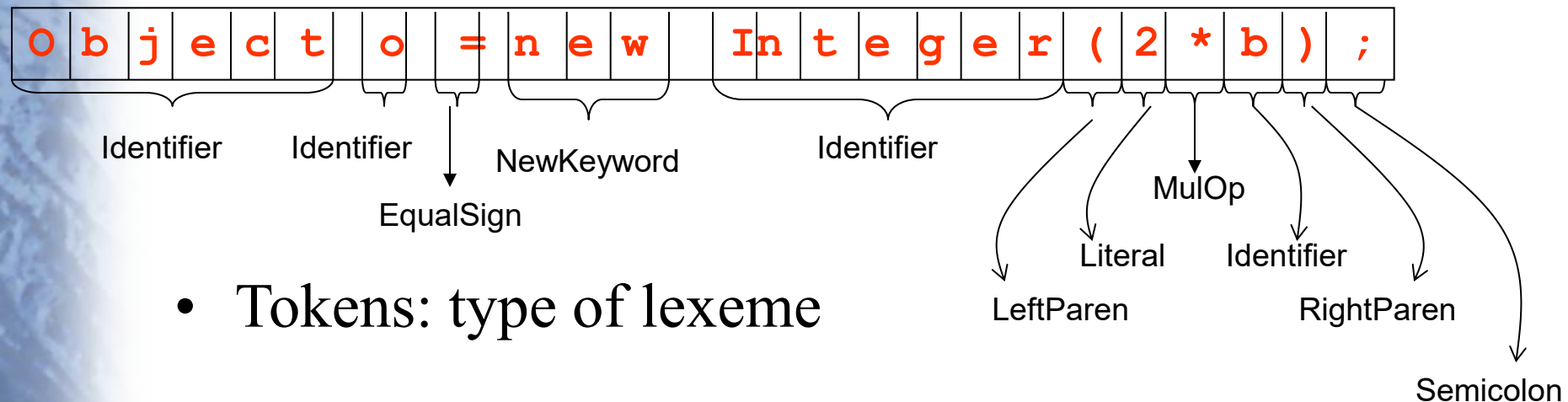
- Reasons to separate lexical and syntax analysis:
 - **Simplicity** - less complex approaches can be used for lexical analysis; separating them simplifies the parser
 - **Efficiency** - separation allows optimization of the lexical analyzer
 - **Portability** - parts of the lexical analyzer may not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

Lexical Analysis

- A lexical analyzer (Scanner) 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**

Lexemes and Tokens

- Lexeme: smallest unit of syntax
 - lexemes identified by lexical analyzers
 - e.g.



- Tokens: type of lexeme

Lexical Analyzer (Scanner)

- **Main task: identify tokens**
 - **Basic building blocks of programs**
 - *E.g.* keywords, identifiers, numbers, punctuation marks
- **Desk calculator language example:**
 - read A**
 - sum := A + 3.45e-3**
 - write sum**
 - write sum / 2**

Formal definition of tokens

- A set of tokens is a set of strings over an alphabet
 - {read, write, +, -, *, /, :=, 1, 2, ..., 10, ..., 3.45e-3, ...}
- A set of tokens is a *regular set* that can be defined by comprehension using a *regular expression*
- For every regular set, there is a *deterministic finite automaton* (DFA) that can recognize it
 - (Aka deterministic Finite State Machine (FSM))
 - *i.e.* determine whether a string belongs to the set or not
 - Scanners extract tokens from source code in the same way DFAs determine membership

Regular Expressions

A regular expression (RE) is:

- A single character
- The empty string, ϵ
- The concatenation of two regular expressions
 - *Notation:* $RE_1 RE_2$ (*i.e.* RE_1 followed by RE_2)
- The union of two regular expressions
 - *Notation:* $RE_1 | RE_2$
- The closure of a regular expression
 - *Notation:* RE^*
 - $*$ is known as the *Kleene star*
 - $*$ represents the concatenation of 0 or more strings
- **Caution:** notations for regular expressions vary
 - Learn the basic concepts and the rest is just syntactic sugar

Lexical Analysis

- 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
- We only discuss approach 2

Lexical Analysis

- 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

- 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

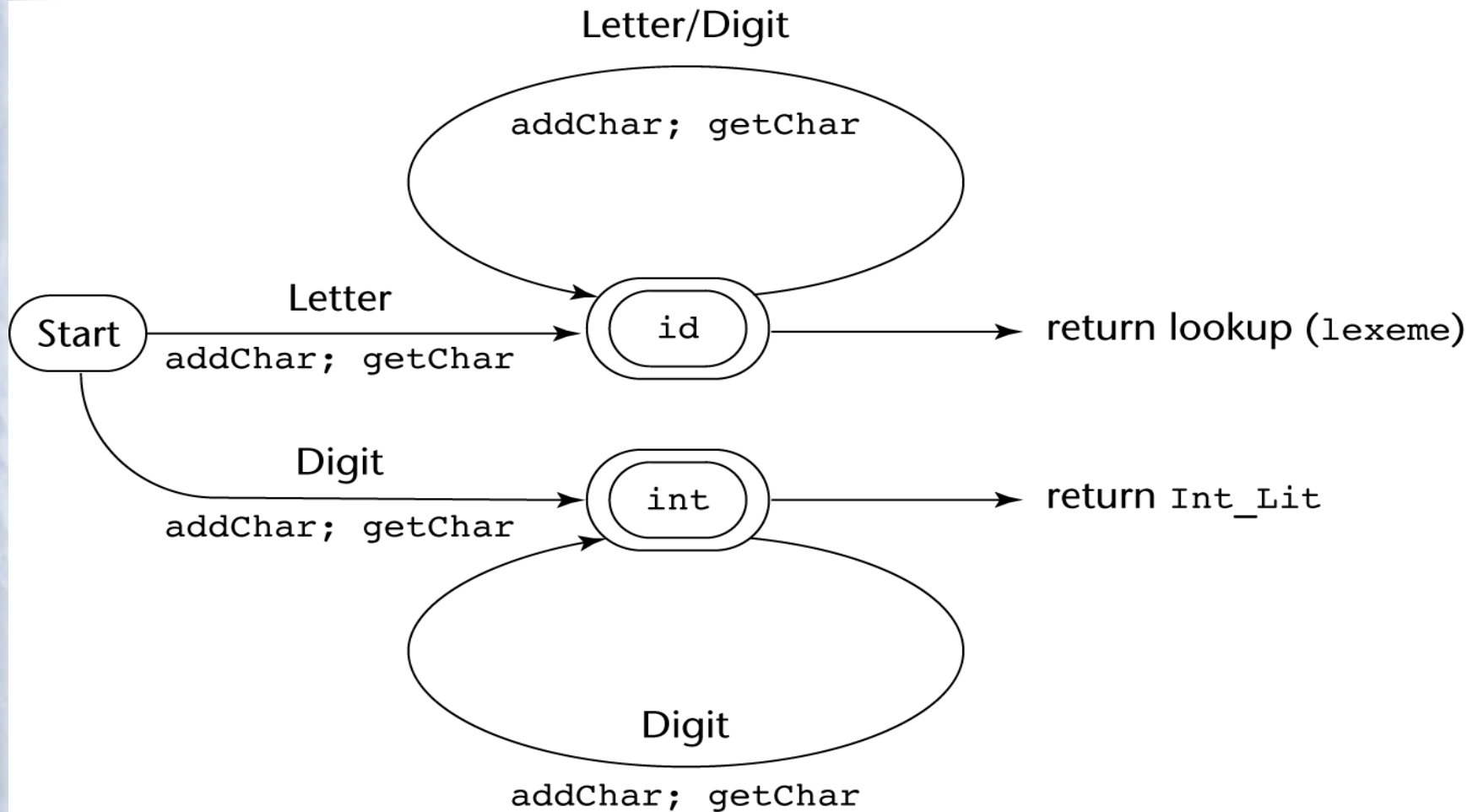
Lexical Analysis

- 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

- 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

- 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

...

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

Lexical Analyzer

Implementation:

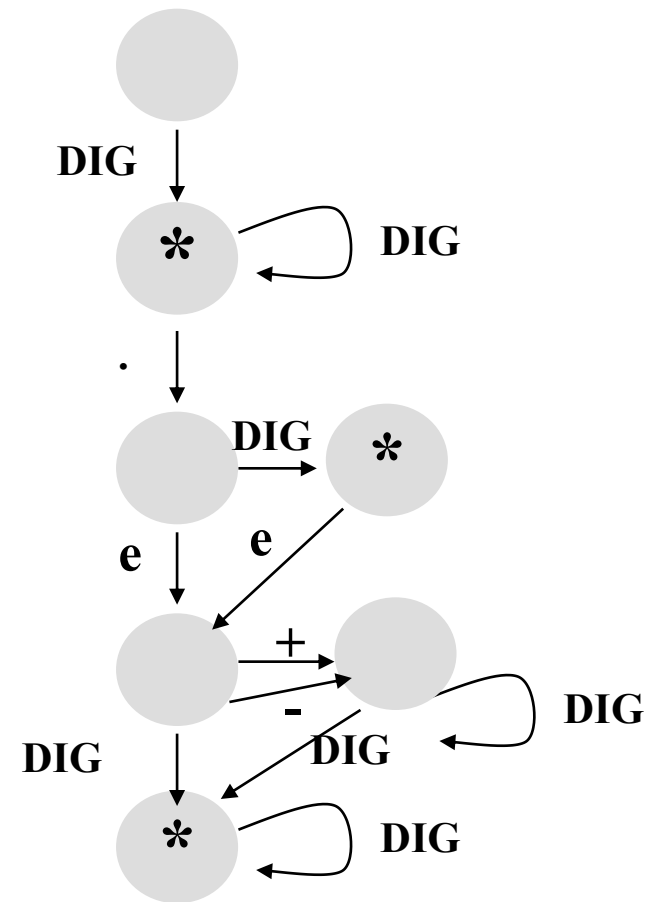
→ SHOW `front.c` (pp. 172–177)

- Following is the output of the lexical analyzer of `front.c` when used on `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (  
Next token is: 11 Next lexeme is sum  
Next token is: 21 Next lexeme is +  
Next token is: 10 Next lexeme is 47  
Next token is: 26 Next lexeme is )  
Next token is: 24 Next lexeme is /  
Next token is: 11 Next lexeme is total  
Next token is: -1 Next lexeme is EOF
```

Token Definition Example

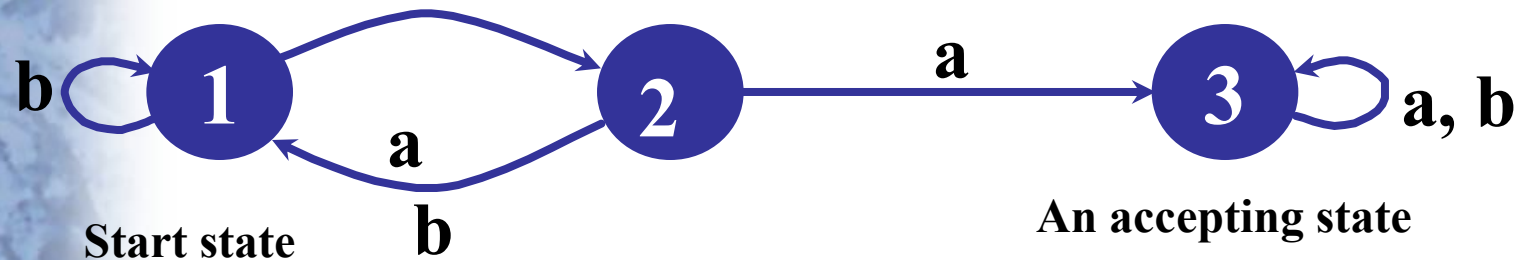
- Numeric literals in Pascal, e.g.
1, 123, 3.1415, 10e-3, 3.14e4
- Definition of token *unsignedNum*
 $DIG \rightarrow 0|1|2|3|4|5|6|7|8|9$
 $unsignedInt \rightarrow DIG\ DIG^*$
 $unsignedNum \rightarrow$
 $unsignedInt$
 $((. unsignedInt) | \epsilon)$
 $((e (+ | - | \epsilon) unsignedInt) | \epsilon)$
- Notes:
 - Recursion is not allowed!
 - Parentheses used to avoid ambiguity
 - It's always possible to rewrite removing epsilons



- FAs with epsilons are nondeterministic.
- NFAs are much harder to implement (use backtracking)
- Every NFA can be rewritten as a DFA (gets larger, though)

Simple Problem

- Write a C program which reads in a character string, consisting of a's and b's, one character at a time. If the string contains a double aa, then print string accepted else print string rejected.
- An abstract solution to this can be expressed as a DFA



The state transitions of a DFA can be encoded as a table which specifies the new state for a given current state and input

		<i>input</i>	
		a	b
<i>current state</i>	1	2	1
	2	3	1
	3	3	3

```
#include <stdio.h>
```

```
main()
```

```
{ enum State {S1, S2, S3};
```

```
enum State currentState = S1;
```

```
int c = getchar();
```

```
while (c != EOF) {
```

```
    switch(currentState) {
```

```
        case S1: if (c == 'a') currentState = S2;
```

```
                if (c == 'b') currentState = S1;
```

```
                break;
```

```
        case S2: if (c == 'a') currentState = S3;
```

```
                if (c == 'b') currentState = S1;
```

```
                break;
```

```
        case S3: break;
```

```
    }
```

```
    c = getchar();
```

```
}
```

```
if (currentState == S3) printf("string accepted\n");
```

```
else printf("string rejected\n");
```

```
}
```

an approach in C

Using a table simplifies the program

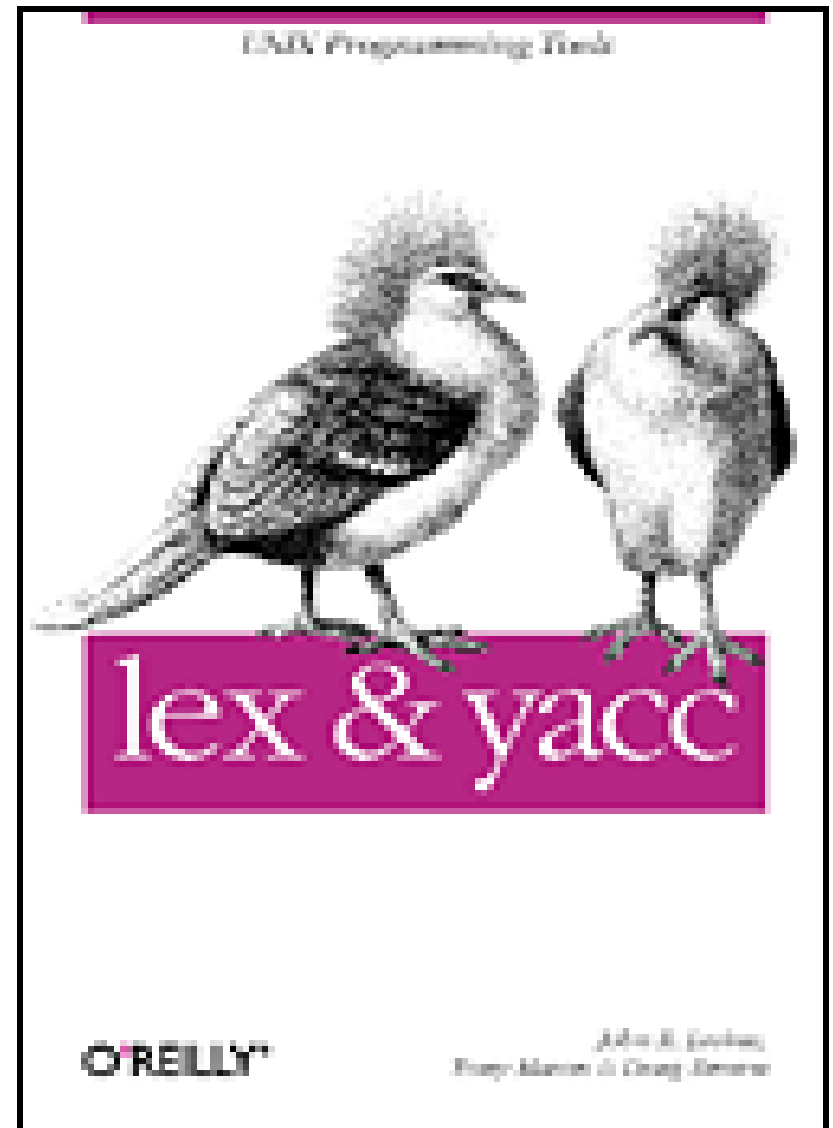
```
#include <stdio.h>
main()
{ enum State {S1, S2, S3};
  enum Label {A, B};
  enum State currentState = S1;
  enum State table[3][2] = {{S2, S1}, {S3, S1}, {S3, S3}};
  int label;
  int c = getchar();
  while (c != EOF) {
    if (c == 'a') label = A;
    if (c == 'b') label = B;
    currentState = table[currentState][label];
    c = getchar();
  }
  if (currentState == S3) printf("string accepted\n");
  else printf("string rejected\n");
}
```

Lex

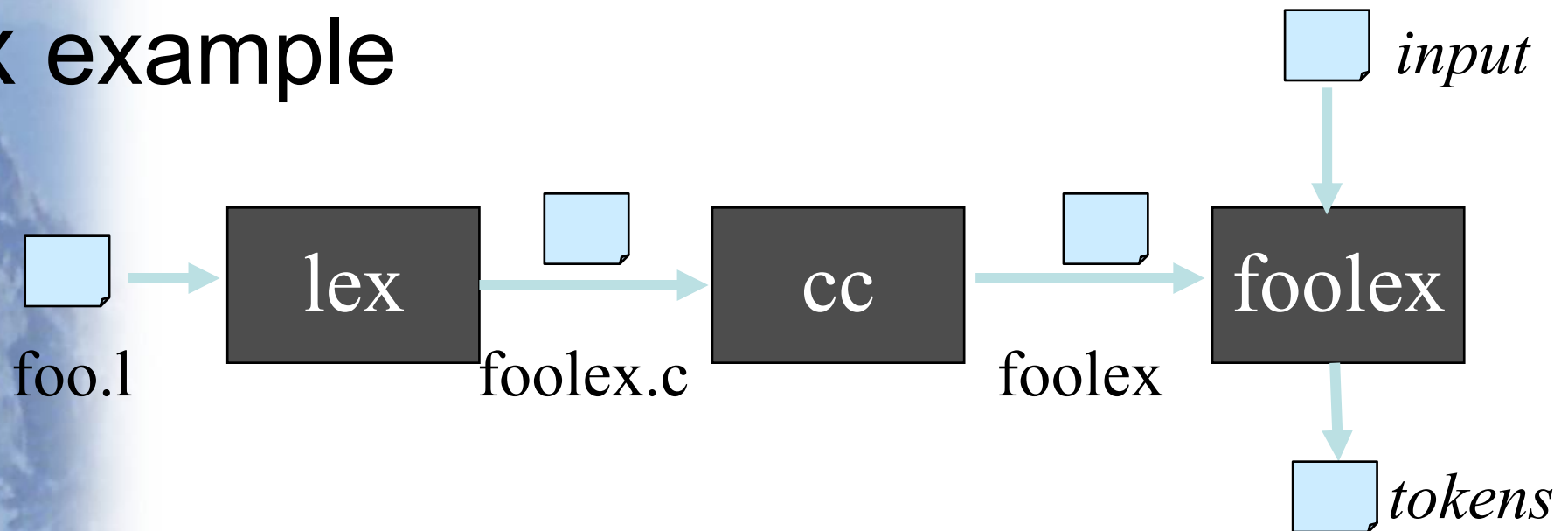
- **Lexical analyzer generator**
 - **It writes a lexical analyzer**
- **Assumption**
 - **each token matches a regular expression**
- **Needs**
 - **set of regular expressions**
 - **for each expression an action**
- **Produces**
 - **A C program**
- **Automatically handles many tricky problems**
- **flex is the gnu version of the venerable unix tool lex.**
 - **Produces highly optimized code**

Scanner Generators

- E.g. lex, flex
- These programs take a table as their input and return a program (*i.e.* a scanner) that can extract tokens from a stream of characters
- A very useful programming utility, especially when coupled with a parser generator (e.g., yacc)
- standard in Unix



Lex example



```
> flex -ofoolex.c foo.1
> cc -ofoolex foolex.c -lfl
```

```
>more input
begin
  if size>10
    then size * -3.1415
  end
```

```
> foolex < input
Keyword: begin
Keyword: if
Identifier: size
Operator: >
Integer: 10 (10)
Keyword: then
Identifier: size
Operator: *
Operator: -
Float: 3.1415 (3.1415)
Keyword: end
```

A Lex Program

```
... definitions ... DIG [0-9]
                    ID [a-z][a-z0-9]*
%%
{DIG}+             printf("Integer\n");
... rules ...      {DIG}+"."{DIG}* printf("Float\n");
                    {ID}           printf("Identifier\n");
%%                [ \t\n]+         /* skip whitespace */
... subroutines ... . printf("Huh?\n");
                    %%
                    main(){yylex();}
```

RE Syntax

Flex's RE syntax

x	character 'x'
.	any character except newline
[xyz]	<i>character class</i> , in this case, matches either an 'x', a 'y', or a 'z'
[abj-oZ]	<i>character class</i> with a range in it; matches 'a', 'b', any letter from 'j' through 'o', or 'Z'
[^A-Z]	<i>negated character class</i> , i.e., any character but those in the class, e.g. any character except an uppercase letter.
[^A-Z\n]	any character EXCEPT an uppercase letter or a newline
r*	zero or more r's, where r is any regular expression
r+	one or more r's
r?	zero or one r's (i.e., an optional r)
{name}	expansion of the "name" definition (see above)
"[xy]"foo"	the literal string: '[xy]"foo' (note escaped “)
\x	if x is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \x. Otherwise, a literal 'x' (e.g., escape)
rs	RE r followed by RE s (e.g., concatenation)
r s	either an r or an s
<<EOF>>	end-of-file

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

- Two categories of parsers
 - **Top down** - produce the parse tree, beginning at the root
 - Order is that of a leftmost derivation
 - **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

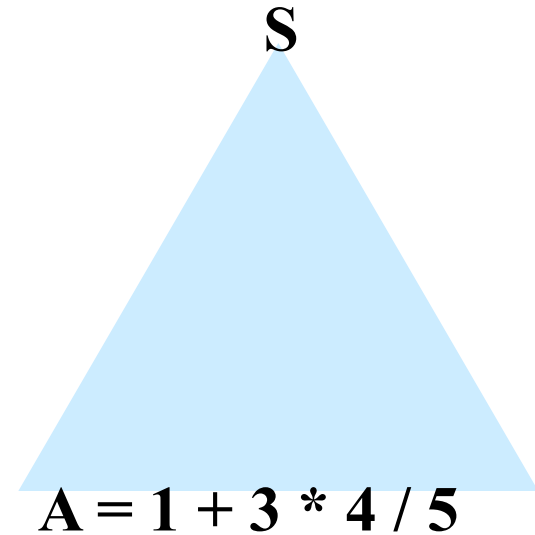
- Top-down Parsers
 - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
 - Recursive descent - a coded implementation
 - LL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

The Parsing Problem

- 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

Top down vs. bottom up parsing

- The parsing problem is to connect the root node S with the tree leaves, the input
- **Top-down parsers:** starts constructing the parse tree at the top (root) of the parse tree and move down towards the leaves. Easy to implement by hand, but work with restricted grammars.
examples:
 - Predictive parsers (e.g., LL(k))
- **Bottom-up parsers:** build the nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handle a larger class of grammars.
examples:
 - shift-reduce parser (or LR(k) parsers)
- Both are general techniques that can be made to work for all languages (but not all grammars!).



Parsing complexity

- How hard is the parsing task?
 - Parsing an arbitrary Context Free Grammar is $O(n^3)$, e.g., it can take time proportional the cube of the number of symbols in the input. This is bad!
 - If we constrain the grammar somewhat, we can always parse in linear time. This is good!
 - 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)
 - Linear-time parsing
 - LL parsers
 - Recognize LL grammar
 - Use a top-down strategy
 - LR parsers
 - Recognize LR grammar
 - Use a bottom-up strategy
- **LL(n)** : **Left to right, Leftmost derivation, look ahead at most n symbols.**
 - **LR(n)** : **Left to right, Right derivation, look ahead at most n symbols.**

Recursive-Descent Parsing

- Recursive Descent Process
 - 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

- A grammar for simple expressions:

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \}$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}$

$\langle \text{factor} \rangle \rightarrow \text{id} \mid (\langle \text{expr} \rangle)$

Recursive-Descent Parsing

- 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

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

void expr() {

    /* Parse the first term */

    term();
```

Recursive-Descent Parsing

```
/* 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

- 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

```
/* 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

```
/* 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 (continued)

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

```
Next token is: 25 Next lexeme is (      Next token is: 11 Next lexeme is total
Enter <expr>                          Enter <factor>
Enter <term>                          Next token is: -1 Next lexeme is EOF
Enter <factor>                        Exit <factor>
Next token is: 11 Next lexeme is sum   Exit <term>
Enter <expr>                          Exit <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>
```

Recursive-Descent Parsing

- 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
 - Direct
 - » $A \rightarrow A + B$
 - Indirect
 - » $A \rightarrow B a A$
 - » $B \rightarrow A b$

Recursive-Descent Parsing

- 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: $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$
(If $\alpha \Rightarrow^* \varepsilon$, ε is in $\text{FIRST}(\alpha)$)

Recursive-Descent Parsing

- 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:
 - $A \rightarrow a \mid bB \mid cAb$
 - $A \rightarrow a \mid aB$
 - The FIRST sets for RHSs of these rules are a, b, and c for the first example which are disjoint.
 - For the second example FIRST sets are a, a which are not disjoint.

Recursive-Descent Parsing

- Left factoring can resolve the problem

Replace

$\langle \text{variable} \rangle \rightarrow \text{identifier} \mid \text{identifier} [\langle \text{expression} \rangle]$

with

$\langle \text{variable} \rangle \rightarrow \text{identifier} \langle \text{new} \rangle$

$\langle \text{new} \rangle \rightarrow \varepsilon \mid [\langle \text{expression} \rangle]$

or

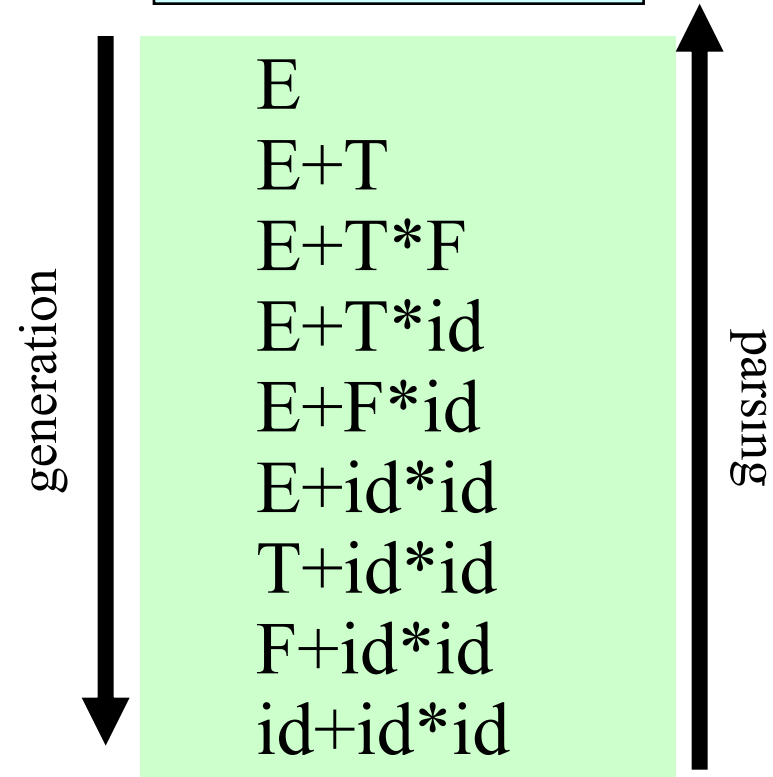
$\langle \text{variable} \rangle \rightarrow \text{identifier} [[\langle \text{expression} \rangle]]$

(the outer brackets are metasymbols of EBNF)

Bottom-up Parsing

- Recall the definition of a derivation and a rightmost derivation.
- Each of the lines is a (right) sentential form
- 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

(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



Handles

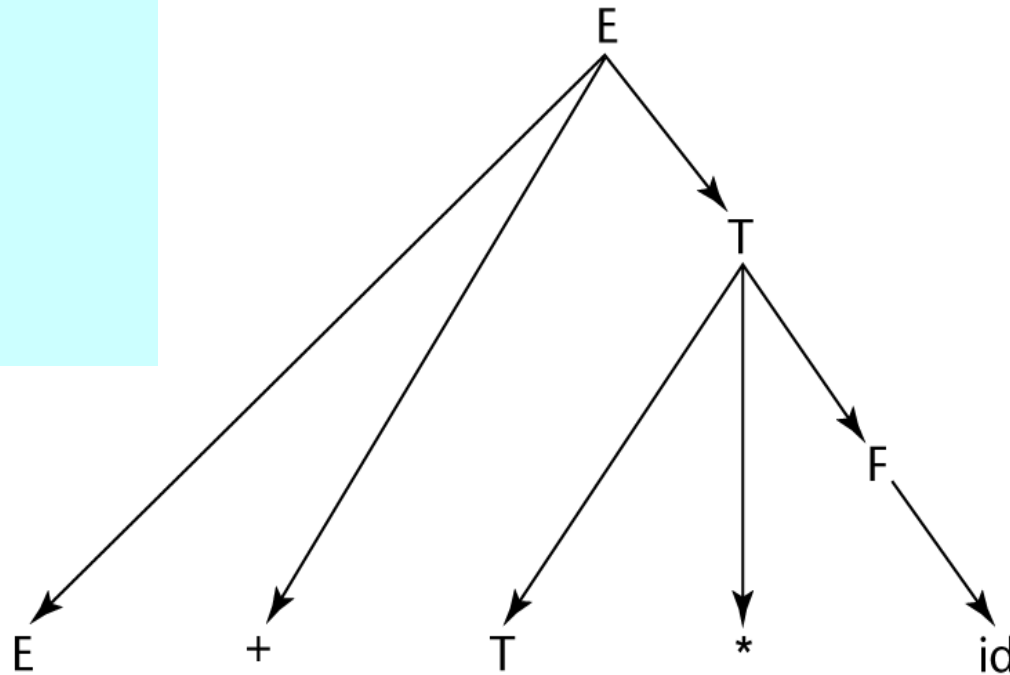
- Intuition: A handle of a string s is a substring a such that :
 - a matches the RHS of a production $A \rightarrow a$; and
 - replacing a by the LHS A represents a step in the reverse of a rightmost derivation of s .
- Example : Consider the grammar
$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$
- The rightmost derivation for the input $abbcd$ is
$$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$$
- The string $aAbcde$ can be reduced in two ways:
 - (1) $aAbcde \Rightarrow aAde$; and
 - (2) $aAbcde \Rightarrow aAbcBe$
- But (2) isn't a rightmost derivation, so Abc is the only handle.
- Note: the string to the right of a handle will only contain non-terminals

Phrases, simple phrases and handles

- Def: β is the *handle* of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta w$
- 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 $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$
- 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

Phrases, simple phrases and handles

$E \rightarrow E+T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow id$



Bottom-up Parsing

- 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

- 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

- 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

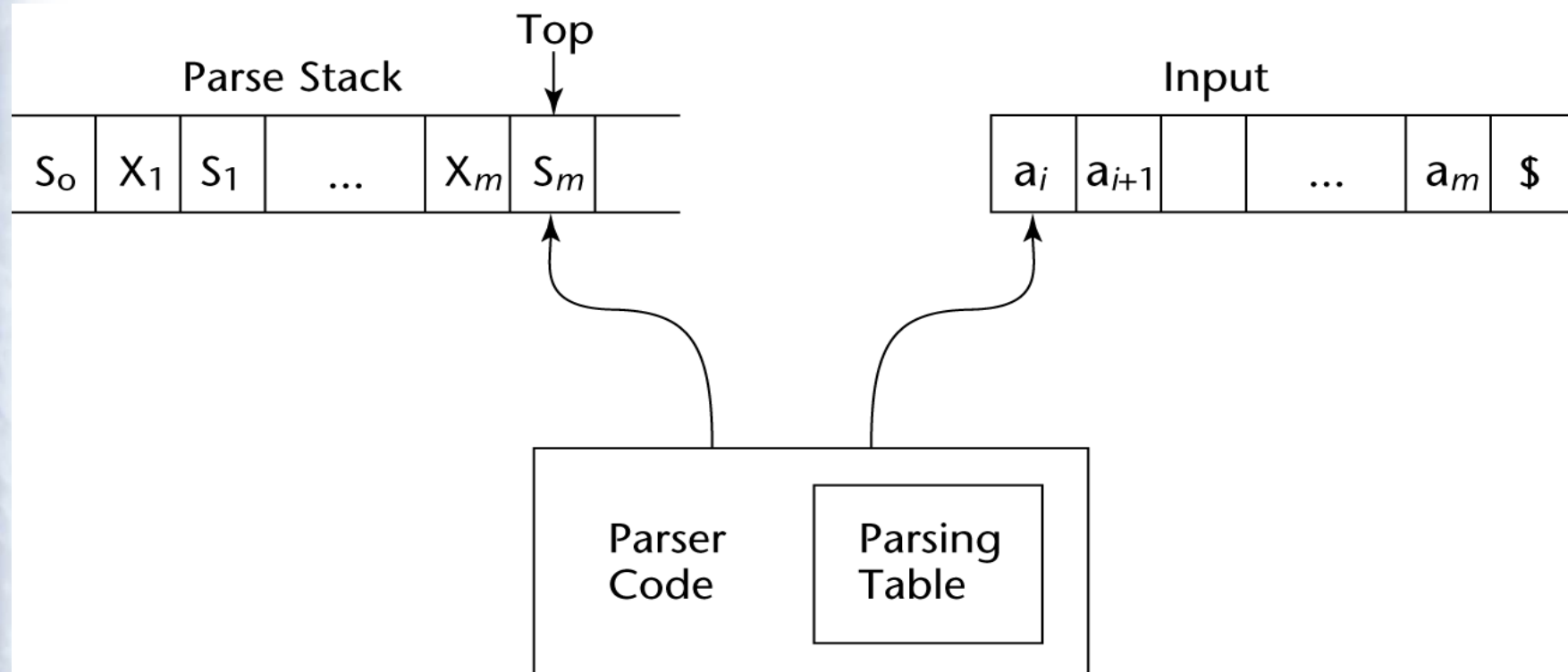
- An LR configuration stores the state of an LR parser

$(\textcolor{red}{S_0 X_1 S_1 X_2 S_2 \dots X_m S_m}, \textcolor{blue}{a_i a_{i+1} \dots a_n \$})$
STACK INPUT

Bottom-up Parsing

- 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

- Initial configuration: $(S_0, a_1 \dots a_n \$)$
- Parser actions:
 - If $\text{ACTION}[S_m, a_i] = \text{Shift } S$, the next configuration is:
 $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$
 - If $\text{ACTION}[S_m, a_i] = \text{Reduce } A \rightarrow \beta$ and $S = \text{GOTO}[S_{m-r}, A]$, where $r = \text{length of } \beta$, the next configuration is
 $(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} AS, a_i a_{i+1} \dots a_n \$)$

Bottom-up Parsing

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

S:Shift

LR Parsing Table

Action entries

Sx = Shift and go to state x

Rx = Reduce using production rule no y

GOTO transitions used after reductions.

(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**

R: Reduce

State	Action						Goto		
	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			

state

Parsing Process

Next token

Go to state 3

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

Bottom-up Parsing

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

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