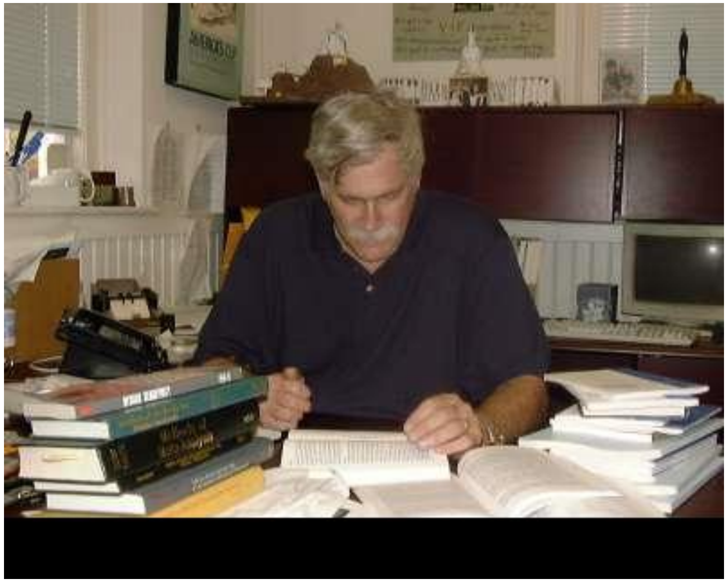
Chapter 4: Lexical and Syntax Analysis

CMSC 124, 1st Semester, AY 2009-10



Syntactic Elements of a Language

1. Character Set
2. Identifiers
3. Operator Symbols
4. Keywords and Reserved Words
5. Noise Words
6. Comments
7. Blanks (Spaces)
8. Delimiters and Brackets
9. Free-Field and Fixed-Field formats
10. Expressions
11. Statements

Chapter 4a: Lexical Analysis

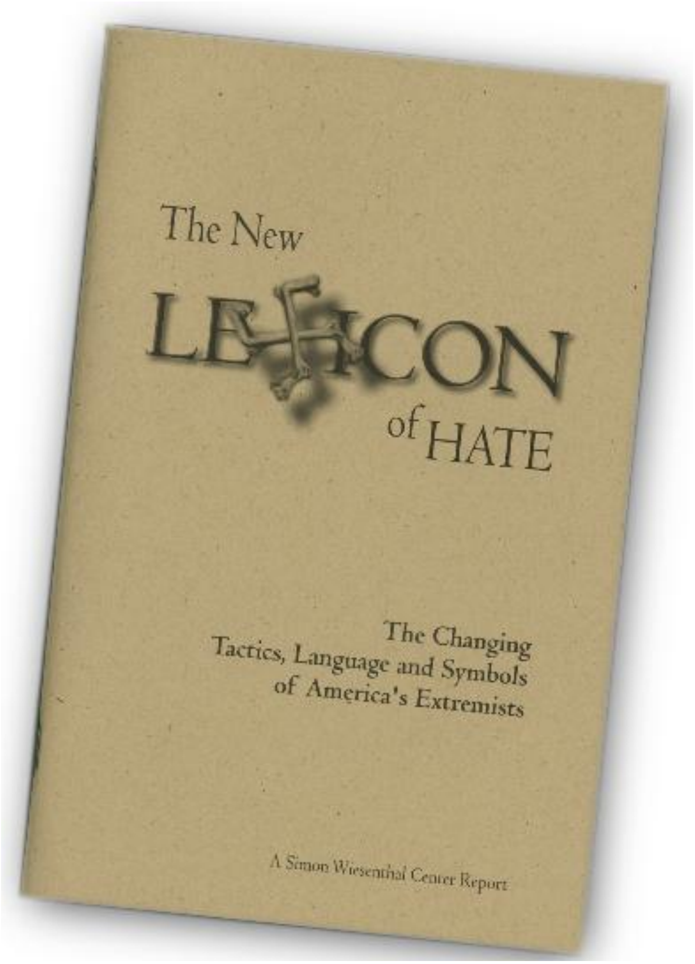
Review from Chapter 3 Slide

- AKA linear analysis/scanning.

- It collects characters into logical groupings and assign internal groupings according to their structure.

- Character groupings are called **lexemes.**

- Internal codes are called **tokens.**



In-Depth: Lexical Analysis

Primary Task

• Reading the input characters and producing as output: Sequence of tokens that the parser uses for syntax analysis.

Secondary Tasks

• Stripping out comments and white spaces.

• Correlating error messages.

Take Note!

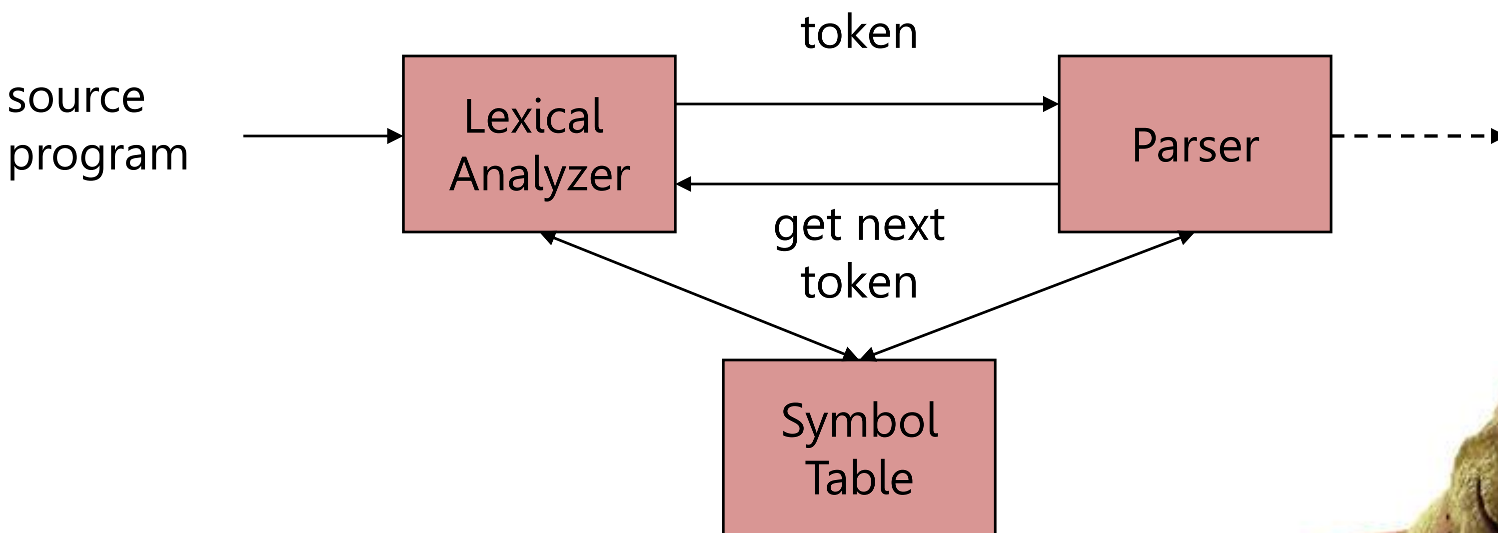
• Lexical analyzer reads the program character-by-character.

• There are times to look ahead several characters beyond the lexeme for a pattern before a match can be announced.

Chapter 4a: Lexical Analysis

In-Depth: Lexical Analysis

A **lexical analyzer** is commonly implemented as subroutine or a coroutine of the parser.
It is called by the parser when it needs a new token.



Tokens, Patterns and Lexemes

Tokens

- Terminal symbols in the grammar.
  - Keywords, operators, identifiers, constants, literal strings, punctuation symbols (parentheses, commas and semicolons)

Lexemes

- Sequences of input characters that comprises a single token

Patterns

- Rules to describe the set of lexemes that can represent a particular token in source programs

Chapter 4a: Lexical Analysis

Tokens, Patterns and Lexemes

Consider this: if a > b mygrade = 0 * 99

TOKEN	LEXEME	INFORMAL PATTERN DESC
IDENT (identifier)	a, b, mygrade	A letter followed by digits and letters.
REL_OP (relational operator)	>	< | <= | <> | == | >= | >
IF (if keyword)	if	if
MULT_OP (multiplicat'n operator)	*	*
INT_LIT (integer)	0, 99	Combination of any numeric constants.
ASSIGN_OP (assignment operator)	=	=

Attribute for Tokens

- Tokens actually influence parsing decisions.
- Attributes influence the translation of tokens.
- Usually, a token has only one attribute – pointer to the symbol table entry (which contains info about the lexeme kept).

Attribute for Tokens: Sample Table

Lexeme	Token	Attribute-Value
if	IF	-
then	THEN	-
else	ELSE	-
(any valid identifier)	IDENT	pointer to table entry
(any integer)	INT_LIT	value of the integer
<	REL_OP	LT
<=	REL_OP	LE
=	REL_OP	EQ
< >	REL_OP	NE
>	REL_OP	GT
>=	REL_OP	GE

Chapter 4a: Lexical Analysis

Attribute for Tokens

Consider this: if a > b mygrade = 0 * 99

The attributes are written as a sequence of pairs.

```
< IF,  >
< IDENT, pointer to symbol table entry for a >
< REL_OP, GT >
< IDENT, pointer to symbol table entry for b >
< IDENT, pointer to symbol table entry for mygrade >
< ASSIGN_OP,  >
< INT_LIT, integer value 0 >
< MULT_OP,  >
< INT_LIT, integer value 99 >
```

Chapter 4a: Lexical Analysis

Recognition of Tokens

- An approach in building a lexical analyzer:
  - ✓ Design a state transition diagram that describe the token patterns of the language, and
  - ✓ Write a program that implements the diagram.

- A **state (transition) diagram** is a graph that describe valid token patterns of a certain language.

Assumptions

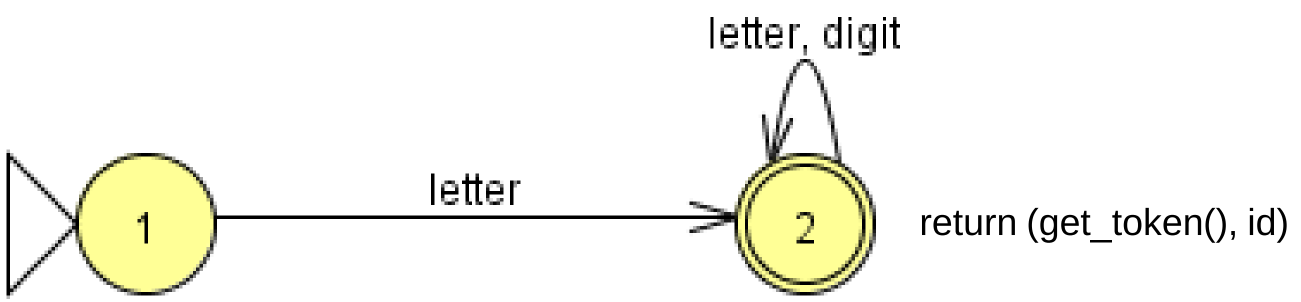
- Keywords are reserved words.
- Integers dealt with are actually unsigned integers.
- Lexemes are separated by white spaces.
  - Blanks, tabs, newlines.
- The chars '==' is used to represent to compare equality.

Chapter 4a: Lexical Analysis

Transition Diagram for Identifiers and Keywords

Remember (from a certain slide before):

TOKEN	SAMPLE LEXEME	INFORMAL PATTERN DESC
IDENT (identifier)	a, b, mygrade	A letter followed by digits and letters.
IF (if keyword)	if	if



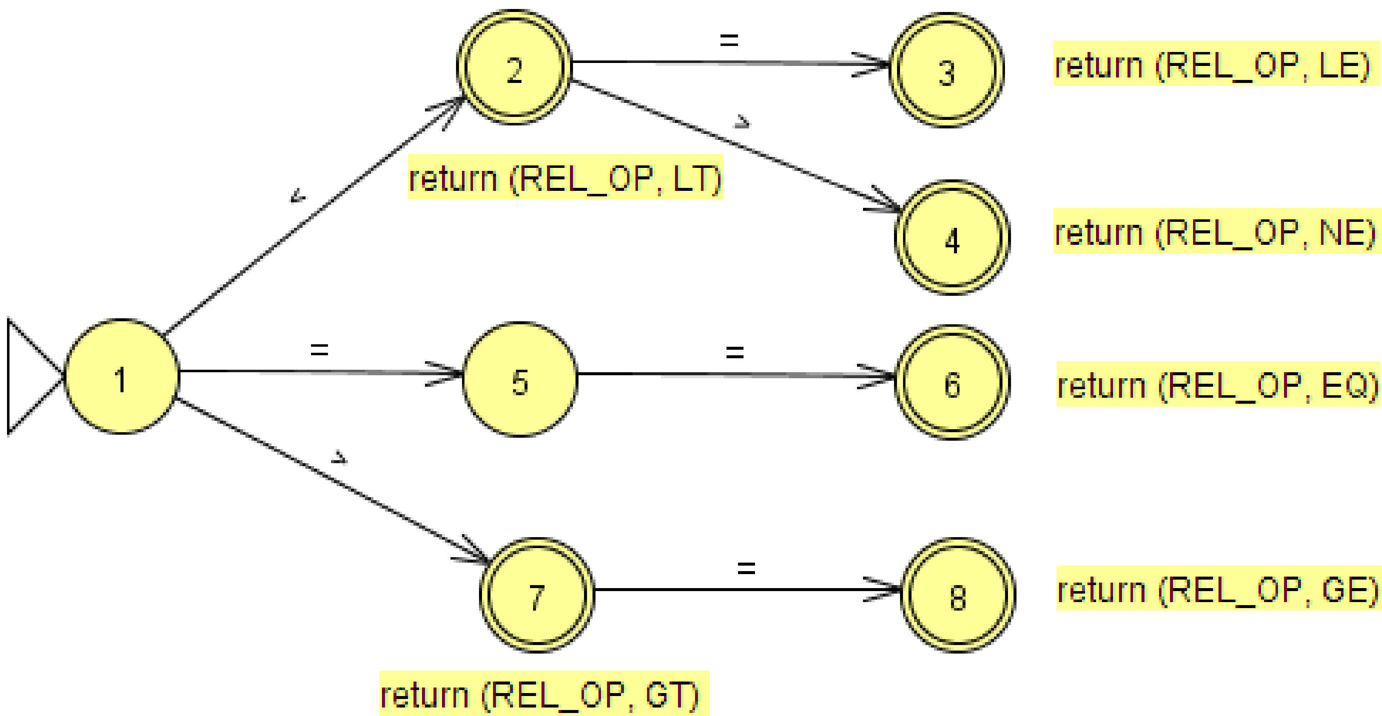
The get_token function may either return **(IDENT, pointer to symbol table entry for that identifier)** or the corresponding token for the keywords.

By the way, what does the get_token function do?

Transition Diagram for Relational Operators

Remember:

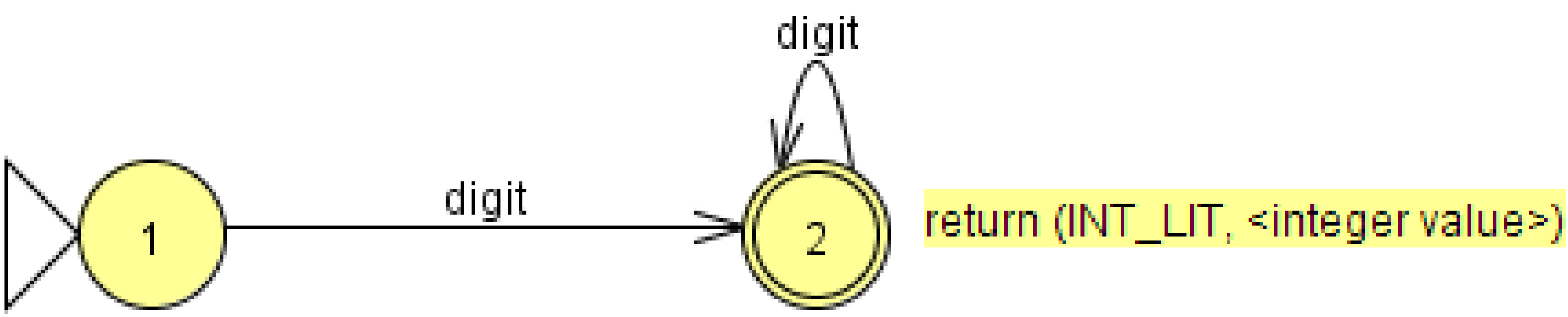
<table>
<tr><th>TOKEN</th><th>SAMPLE LEXEME</th><th>INFORMAL PATTERN DESC</th></tr>
<tr><td>REL_OP
(relational operator)</td><td>></td><td>< | <= | <> | == | >= | ></td></tr>
</table>



Transition Diagram for Integers

Remember:

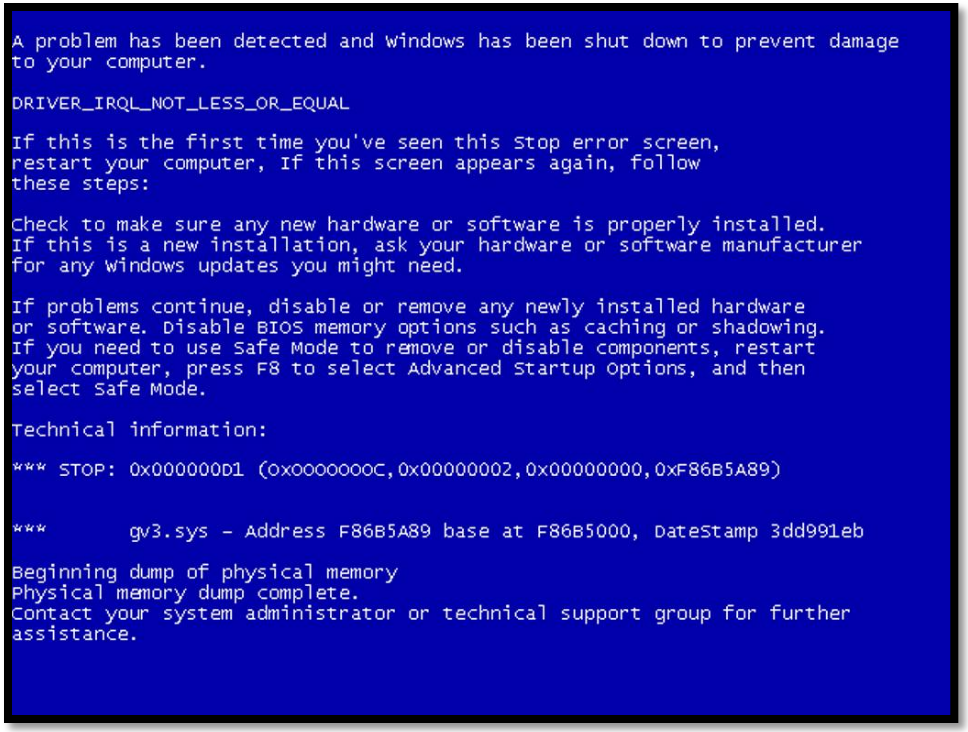
TOKEN	SAMPLE LEXEME	INFORMAL PATTERN DESC
INT_LIT (integer)	0, 99	Combination of any numeric constants.



Chapter 4a: Lexical Analysis

Detection: Possible Errors

- Illegal characters
- String forming name of identifier too long
- Numeric value too large or too small
- Comments not closed properly



Something to Ponder

Can you design the transition diagram of comments (enclosed in /* */) ?

Something to Ponder

Chapter 4b: Syntax Analysis

The Parser, the 'Syntax Analyzer'

Goals of a Parser

- **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 .

Two Categories of Parsers

1. Top-Down

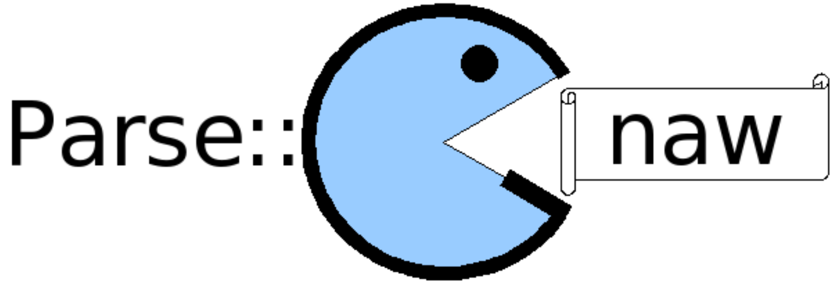
- Produce the parse tree, beginning at the root.
- Order is that of a **leftmost derivation**.

2. Bottom-Up

- Produce the parse tree, beginning at the leaves.
- Order is that of the **reverse of a rightmost derivation**.

Note

Parsers look only one token ahead in the input.



Notational Conventions

- **Terminal Symbols**
  Lowercase letters at the beginning of the alphabet (a, b, c…).
- **Nonterminal Symbols**
  Uppercase letters at the beginning of the alphabet (A, B, C…).
- **Terminals or Nonterminals**
  Uppercase letters at the end of the alphabet (W, X, Y, Z).
- **Strings of Terminals**
  Lowercase letters at the end of the alphabet (w, x, y, z).
- **Mixed Strings (Terminals and/or Nonterminals)**
  Lowercase Greek letters (α, β, δ, γ).

Top-Down Parsers

Given a sentential form, **xAa**, choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A .

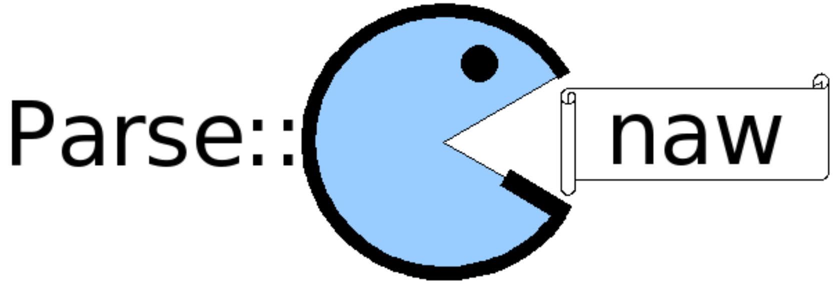
Example

A-rules: A->bB, A -> cBb, and A->a

Which one should be chosen?

Possible next sentential forms:

- ✓ xbBa
- ✓ xcBba
- ✓ xaa



Top-Down Parsers: Most Common TD Parsing Algorithms

- **Recursive Descent**
  A coded implementation.

- **LL Parsers**
  A table-driven implementation.

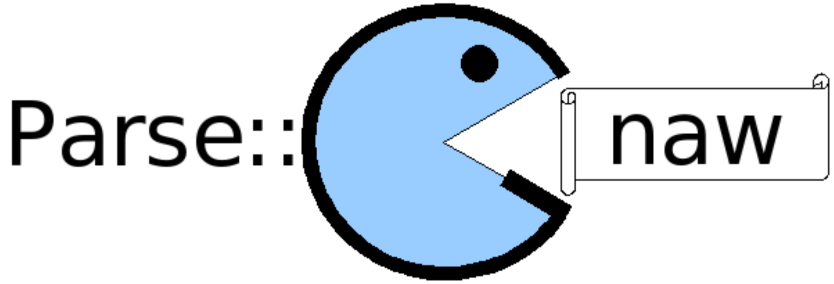
Note
Both are called LL algorithms, with equal power.
- **First L** – left-to-right scan of inputs.
- **Second L** – leftmost derivation.

Given a right sentential form, **a**, what substring of a 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.

Note

The most common BU parsing algorithms are in the LR family.

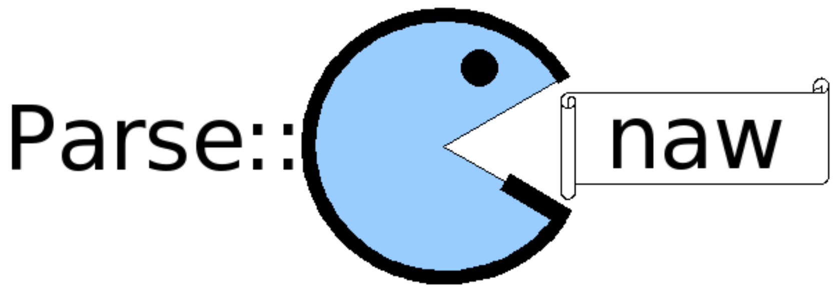
- L – left-to-right scan of inputs.
- R – rightmost derivation.



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.



Chapter 4b: Syntax Analysis

Top-Down: Recursive-Descent Parsing

The Process

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

Extended BNF

- It uses braces and brackets.
- Braces indicate that anything enclosed by them can be omitted or repeated.
- Brackets indicate option.

Example

- <if_stmt> -> if
  <logic_expr><stmt>
  [else <stmt>]
- <ident_list> -> ident {, ident}

Top-Down: Recursive-Descent Parsing

- Given a grammar for simple expressions:
  <expr> ::= <term> { (+ | -) <term> }
  <term> ::= <factor> { (* | /) <factor> }
  <factor> ::= id | (<expr>)

- Assume we have a lexical analyzer named lex, which puts the next token code in **nextToken** (which is a global variable).

Recursive-Descent Parsing: The Process for One RHS

- For each **terminal** symbol in the RHS,
  - Compare it with the next input token.
  - If they match, continue; otherwise, error.

- For each **nonterminal** symbol in the RHS,
  - Call its associated parsing subprogram.

Recursive-Descent Parsing: The Process for One RHS

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

void expr() {
/* 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 == PLUS_CODE ||
         nextToken == MINUS_CODE){
    lex();
    term();
  }
}
```

Top-Down Recursive-Descent Parsing

- The previous routine does not detect errors.

- **Convention**
  - ✓ Every parsing routine leaves the next token in **nextToken.**
  - ✓ So, whenever a parsing function begins, it is assured that nextToken has the leftmost token of the input not yet used in the parsing process.

Recursive-Descent Parsing: The Process for Multiple RHS

- 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, syntax error.

Chapter 4b: Syntax Analysis

Recursive-Descent Parsing: The Process for Multiple RHS

```
/* 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();
 /* 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 */
 }
```

Something to Ponder

One thing, it is not left recursive.

A recursive-descent parser can be easily written if an appropriate grammar is available.
The key now is what defines an "appropriate grammar" for recursive-descent parsing.

So, what defines an "appropriate grammar"?

Another thing, it has pairwise disjointness.

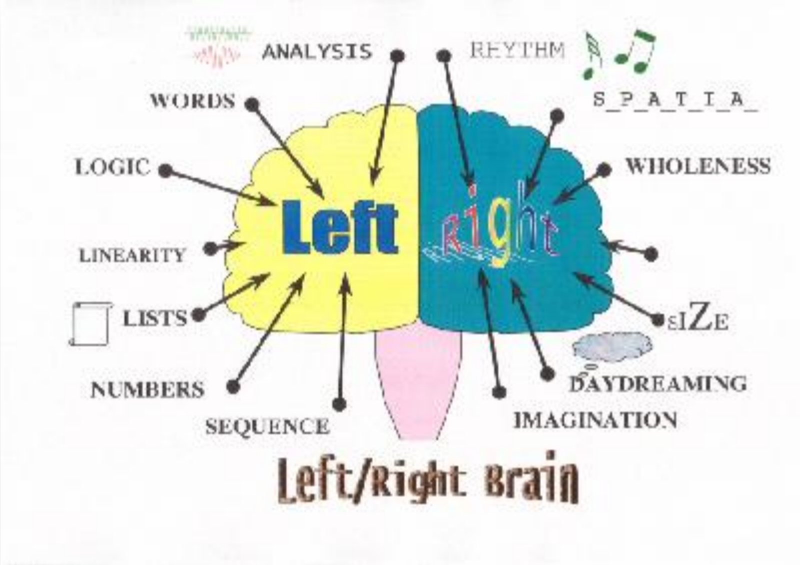
Something to Ponder

Chapter 4b: Syntax Analysis

The LL Grammar Class

1. The Left Recursion Problem

- If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser.
- **Solution:** There is an algorithm to modify a grammar, removing both direct and indirect left recursion.



- **Example**
  - A -> A + B
  - A -> BaA, B -> Ab

Why is this a left recursion problem?

2. Pairwise Disjointness

- The disjointness can be failed – in the inability to determine the correct RHS on the basis of one token of lookahead.

- **Pairwise Disjointness Test**

For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules,

A -> ai and A -> aj

it must be true that

FIRST(ai) ∩ FIRST(aj) = **Φ**

- In other words, if a nonterminal A has more than one RHS, the first terminal symbol that can be generated in a derivation for each of them must be unique to that RHS

- **Example**

A -> a | bB | cAb **PASSED**

A -> a | aB **FAILED**

Something to Ponder

How do we pass the pairwise disjointness test?

Something to Ponder

Passing the Pairwise DisjointnessTest

- The "failures" can be modified to pass the test.
- Consider the following rules:

  <variable> -> identifier | identifier [<expression>]

- Clearly, the above rules are "5.0 " in the test.

Solution 1: Left Factoring

<variable> -> identifier <new>

<new> -> ε | [<expression>]

Empty string

Solution 2: EBNF Extension

<variable> -> identifier

[[<expression>]]

EBNF Metasymbol

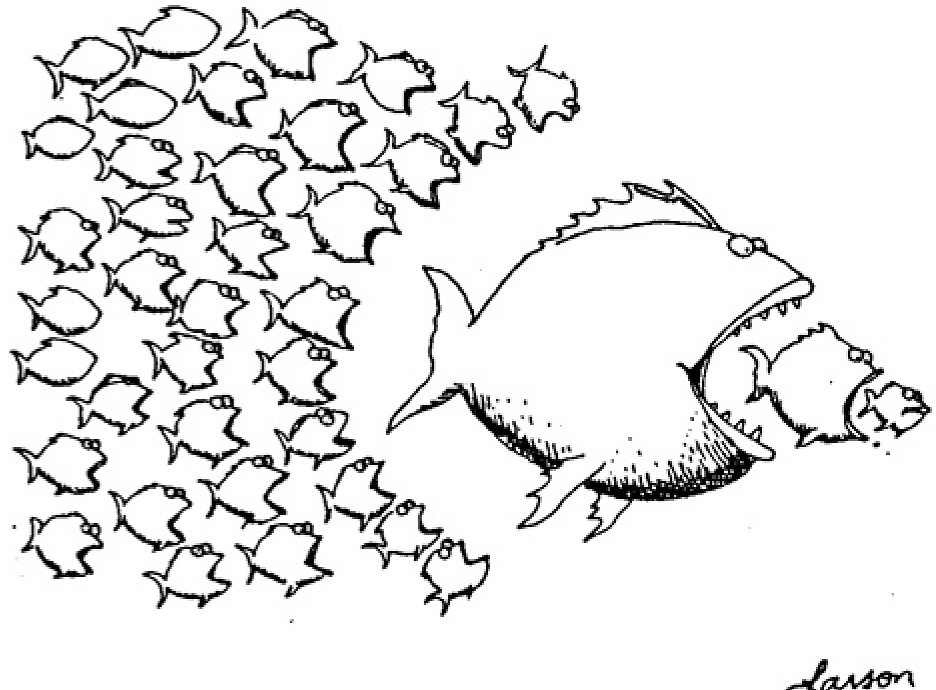
- **Reminder:** Left factoring <u>cannot solve</u> all pairwise disjointness of grammars. There are cases that rules must be rewritten to eliminate the problem.

Chapter 4b: Syntax Analysis

Bottom-Up Parsing

- Left-recursive grammars can be parsed by bottom-up parsers.

- Bottom-up parsers are usually implemented as table driven state machines.



Parsing Problem for Bottom-Up Parsers

Given the ff. grammar:

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

```
E  => E + T
   => E + T * F
   => E + T * id
   => E + F * id
   => E + id * id
   => T + id * id
   => F + id * id
   => id + id * id
```

Parsing Problem for Bottom-Up Parsers

- The underlined part of each sentential form in the derivation is the **RHS** that is rewritten as its **corresponding LHS**.

- The process of bottom-up parsing produces the **reverse** of a rightmost derivation.

- (Based on the derivation)
A bottom-up parser starts with the **last sentential form** (the input sentence) and produces the sequence of sentential forms until all that remains is the start symbol.

The Task of a Bottom-Up Parser

Find specific RHS in the sentential form that must be rewritten to get next (previous) sentential form.

- **Example:** Given the grammar

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

Parse E + T * id

There are 3 possible RHS's: **E+T, T, and id**

But if we select **E+T**, the resulting sentential form would be **E * id** which is illegal!

Parsing Problem for Bottom-Up Parsers

- The correct RHS in a given right sentential form is called the **handle**.

Shift-Reduce Algorithms

- Bottom-up parsers are often called shift-reduce algorithms.

- **Why Shift-Reduce?**
  - ✓ It uses a stack to remember where it has been.
  - ✓ The table contains, for each state and next token:
    - ➢ The next state
    - ➢ Action to be taken
  - ✓ Actions:
    - ➢ Shift – Pushes the token and state onto the stack
    - ➢ Reduce – Pops tokens and states from the stack and produces a phrase.

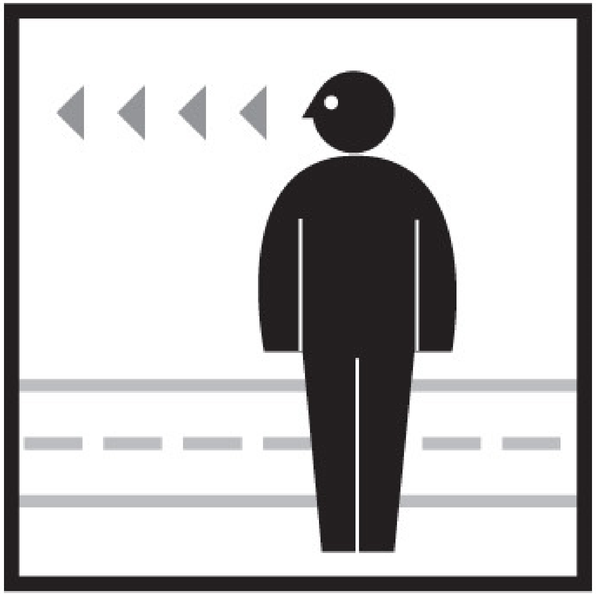
LR Parser

- Parses the source from left to right and constructs a rightmost derivation of the sentence. It is a state machine.

Components

1. **Input String**
   - Stream of tokens (terminals) representing the whole program.

2. **Stack**
   - Contains a list of states it has been in.



LR Parser

- Parses the source from left to right and constructs a rightmost derivation of the sentence.

Components

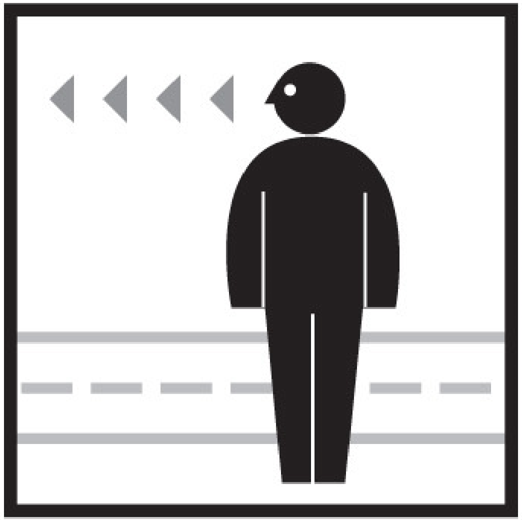
3. **ACTION Table**
   - Specifies the action of the parser, given the parser state and next token.

4. **GOTO Table**
   - Specifies which state to put on top of the parse stack after a reduction is done.

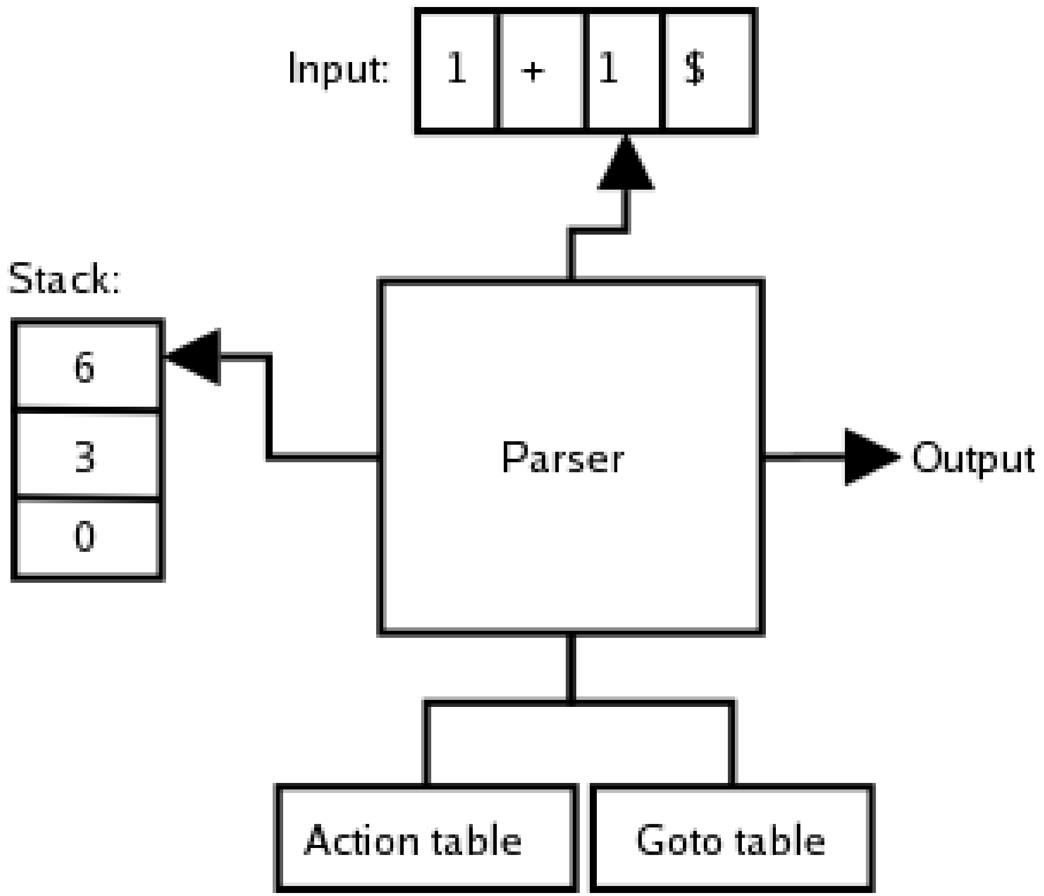
Take Note!

Rows are state names.

Columns are nonterminals.



LR Parser: Architecture

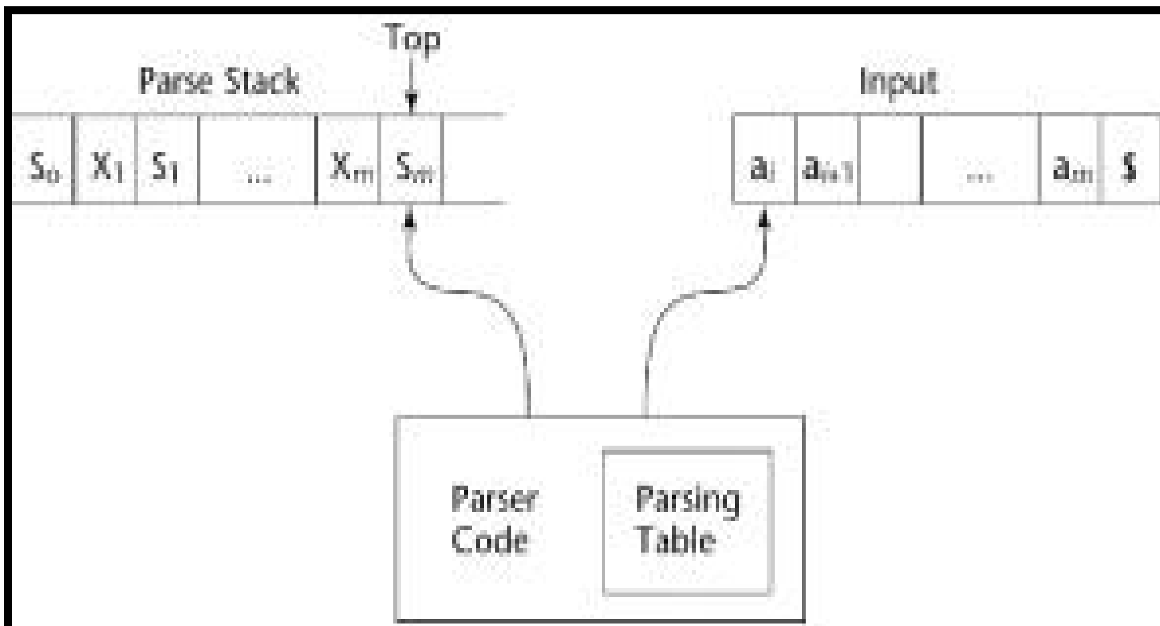


Chapter 4b: Syntax Analysis

LR Parsing Algorithm

The LR parsing algorithm works as follows:

Step 1. The stack is initialized with [**0**]. The current state will always be the state that is at the top of the stack.



LR Parsing Algorithm

Step 2. Given the current state and the current terminal on the input stream an action is looked up in the action table. There are four cases:

- ✓ **a shift s*n***
  - the current terminal is removed from the input stream
  - the state *n* is pushed onto the stack and becomes the current state
- ✓ **a reduce r*m***
  - the number *m* is written to the output stream
  - for every symbol in the right-hand side of rule *m* a state is removed from the stack
  - given the state that is then on top of the stack and the left-hand side of rule *m* a new state is looked up in the goto table and made the new current state by pushing it onto the stack

Chapter 4b: Syntax Analysis

LR Parsing Algorithm

Step 2. Given the current state and the current terminal on the input stream an action is looked up in the action table. There are four cases: (cont.)

- ✓ **an accept** the string is accepted
- ✓ **no action** a syntax error is reported

Step 3. Step 2 is repeated until either the string is accepted or a syntax error is reported.

LR Parsing Algorithm: Concrete Example

state	*action*					*goto*	
	*****	**+**	**0**	**1**	**$**	**E**	**B**
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

ACTION Table

Indexed by a state of the parser and a terminal (including a special terminal $ that indicates the end of the input stream) and contains three types of actions:

- **shift**, which is written as 's*n*' and indicates that the next state is *n*
- **reduce**, which is written as 'r*m*' and indicates that a reduction with grammar rule *m* should be performed
- **accept**, which is written as 'acc' and indicates that the parser accepts the string in the input stream.

Chapter 4b: Syntax Analysis

LR Parsing Algorithm: Concrete Example

state	*action*					*goto*	
	*****	**+**	**0**	**1**	**$**	**E**	**B**
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

GOTO Table
Indexed by a state of the parser and a nonterminal and simply indicates what the next state of the parser will be if it has recognized a certain nonterminal.

LR Parsing Algorithm: Concrete Example

state	*action*					*goto*	
	*****	**+**	**0**	**1**	**$**	**E**	**B**
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

To explain its workings we will use the following small grammar whose start symbol is E:

```
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1
```

Parse the following input:
1 + 1

Quiz::: PARSE 1

Concrete Example

Parsing Procedure for 1+1

State	Input	Output	Stack	Next Action
0	1+1$		[0]	Shift 2
2	+1$		[0,2]	Reduce 5 GOTO[0, B]
4	+1$	5	[0,4]	Reduce 3 GOTO[0, E]
3	+1$	5,3	[0,3]	Shift 6
6	1$	5,3	[0,3,6]	Shift 2
2	$	5,3	[0,3,6,2]	Reduce 5 GOTO[6, B]
8	$	5,3,5	[0,3,6,8]	Reduce 2 GOTO[0, E]
3	$	5,3,5,2	[0,3]	Accept

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.

Notes on LR Parsers

- Algorithms to generate LR parsing tables from given grammars are described in Aho et al (1986).
- A parser table can be generated from a given grammar with a tool, e.g., yacc.