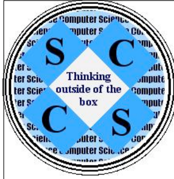
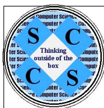


Lexical and Syntax Analyzers: Implementations

Charles R. Hardnett

Copyright © 2003 Spelman College



Lexical Analyzers

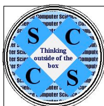
- Regular expression pattern matching programs
  - **Regular expressions use: *, […], + and concatentation**
  - **Regular expressions generate regular languages**
- Matches lexemes in the language and classifies them as tokens
  - **Identifier names**
  - **Keywords/Reserved Words**
  - **Numeric literals**
  - **String literals**
- Match comments and may remove them

Copyright © 2003 Spelman College

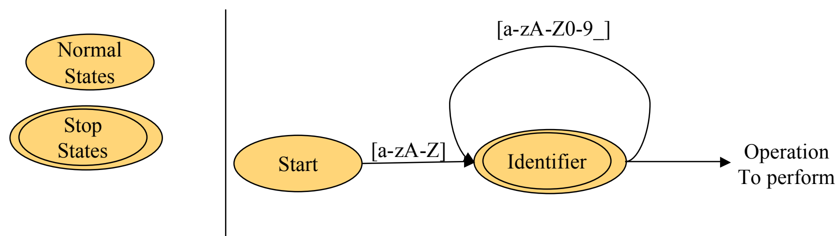
Lexemes vs Tokens

Token (Class)	Lexeme (Instances)
IDENT	Count
ADD_OP	+
IF_KW	if
IDENT	classAvg_1
INT_LIT	5000

- Developing a Lexer is developing Tokens (classes)
  - **Write a formal token description in a specialized language (lex) and generate lexer automatically**
  - **Create state transition diagrams to describe token patterns and write a program**
  - **Create state transition diagrams and construct a table-driven implementation**

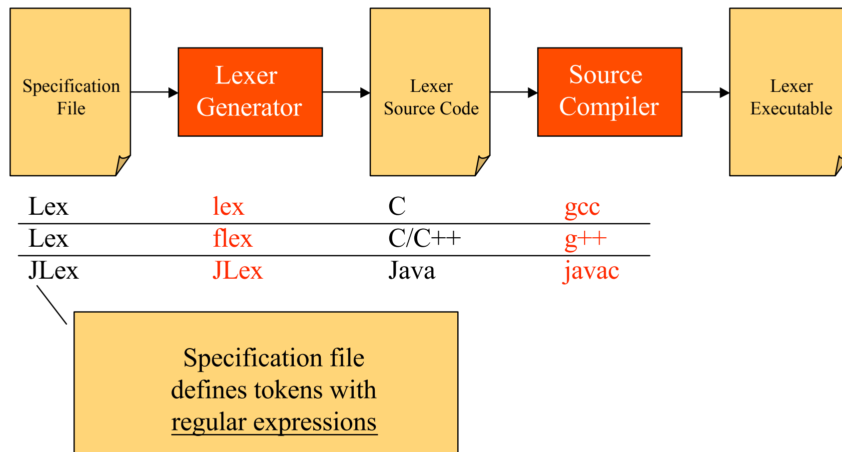


State Diagrams



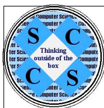
- State diagrams are based on Finite State Machines (FSMs)
  - **Contains states and transitions**
    - **Transitions have input values that cause the transition**
  - **A start state**
  - **1 or more stop states (accept states)**
- Can be implemented using “switch” statement
  - **Each “case” corresponds to type of transition**

Lexer: Implementation with Lex Tools



Lex	lex	C	gcc
Lex	flex	C/C++	g++
JLex	JLex	Java	javac

Specification file defines tokens with regular expressions



Syntax Analyzers

- Implementation is based on the Grammar rules of the language
  - **Context-Free Languages**
- Goals of syntax analyzers
  - **Check the syntax of a given program**
  - **Provide error messages, and continue parsing the program**
- Parsers can be implemented as top-down and bottom-up
- Parsers have look ahead of 0 or 1
  - **Look ahead is the number of tokens the parser examines to decide between rules**

Types of Parsers

- LL Parsers
  - Left-to-right processing of an input sentence
  - Leftmost derivation is generated
- LR Parsers
  - Left-to-right processing of an input sentence
  - Rightmost derivation is generated
- LL(1) Parsers
  - LL Parsers with one token look ahead
- LR(1) Parsers
  - LR Parsers with one token look ahead

Copyright © 2003 Spelman College

LL Grammars

- LL parsing is an intuitive way to parse grammars by-hand i.e. the algorithm we discussed in Chapter 3
- LL parsing can be implemented as Recursive Descent Parsers
- A Recursive Descent Parser
  - A subprogram is created for each NT symbol
  - The body of the subprogram is based on the definition (rule(s)) associated with the NT symbol
  - The recursion is in the fact that subprograms are recursively invoked to handle NT symbols during parsing

Copyright © 2003 Spelman College

A Recursive Descent Parser

```
void expr() {
   expr();

   while (nextTok == PLUS ||
          nextTok == MINUS) {
        lex();
        term();
   }
}
void factor() {
   if (nextTok == ID)
        lex();
   else if (nextTok == LEFT_P) {
        lex();
        expr();
        if (nextTok == RIGHT_P)
            lex();
        else
            error();
    }
   else error();
}
```

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

- The Recursive Descent Parser is derived directly from the grammar
  - **The function calls are correlated with the appearance of NT symbols in the grammar rules**
  - **Lex() is the interface to the lexer to generate the next token (nextTok)**
- What is the problem in the **expr**() routine?

Copyright © 2003 Spelman College

Problems with LL Grammars

```
void expr() {
   expr();

   while (nextTok == PLUS ||
          nextTok == MINUS) {
        lex();
        term();
   }
}
```

- The left-recursion of the <**expr**> rule
  - **The parser will continuously call** expr**()**
- Involves breaking rule into 2 rules

E -> TE'	**T -> FT'**	**F -> id**
E' -> +TE'	**T' -> *FT'**	**| (E)**
| nil	**| nil**	

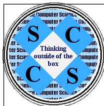
- A second problem
  - **The 1 token lookahead is not enough to allow the parser to make the decision**
    - **<X> -> a<B>**
      | a<X>
- Pairwise Disjointnes test determines when this occurs
  - **Generate FIRST sets for each rule**
  - **FIRST sets are the set of terminal symbols that appear first in each rule**
    - **{a} and {a} in this example**
- Left-Factoring fixes it
  - **X -> a <new>**
  - **<new> -> a<X>**
    | <B>

Copyright © 2003 Spelman College

LR Grammars

- LR Grammars are most widely used grammars for programming languages
- LR Grammars can be parsed with table controlled parsers
- Tools exists to generate these parsers given a LR grammar specification

Copyright © 2003 Spelman College



LR Parse Table

<E> -> <E> + <T>
<E> -> <T>

<T> -> <T> * <F>
<T> -> <F>

<F> -> (<E>)
<F> -> id

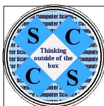
Grammar

	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			

Copyright © 2003 Spelman College

LR Parser Derivation

Stack	Input	Action	Comments
0	id + id$	Shift 5	state 0, id column
0id5	+ id$	Reduce 6	id -> F, F subs for id5 @ state 0
0F3	+ id$	Reduce 4	F -> T, T subs for F3, @ state 0
0T2	+ id$	Reduce 2	T -> E, E subs for T2, @ state 0
0E1	+ id$	Shift 6	state 1, + column
0E1+6	id$	Shift 5	state 6, id column
0E1+6id5	$	Reduce 6	id -> F, F subs for id5, @ state 6
0E1+6F3	$	Reduce 4	F -> T, T subs for F3, @ state 6
0E1+6T2	$	Reduce 2	E + T -> E, E subs E1+6T2, @state 0
0E1	$	accept	



Parser: Implementation with Parser Tools