**Report Submission**

 Lecture Date: 07 Aug 13

 Honey Joshi 11CS10016

 **The objectives of this lecture are**

* To understand how to represent different kinds of patterns.
* Using patterns to detect Lexemes.
* Lexical Analysis using Finite State Automata.

 **Control Flow Diagram for Token Generation**

 **Pattern (var.) Pattern (numbers)**

 **RE Pattern (key)**

 **Lexeme Control Token**

 **Stream of characters**

We define a ‘Toy’ Language

Keywords: if, else, while, do.

Operators: >, <, <=, >=, =

Variable Name: starts with alphabet (letter) then ‘\_’ or numbers.

Numbers: whole numbers, floating point numbers

Define pattern for individual tokens:

* **Keywords**:

Pattern: if, else, while, do.

Token: <if>, <else>, <while>, <do>

No attributes are assigned to the keywords. Only keyword name(id) is given in the token name.

* **Operator**:

Pattern: >, <, <=, >=, =

Token : < operation, attribute >

 < op, GT >

 < op, LT >

 < op, GE >

* **Variable** **Names**:

We use Regular Expressions (RE) to get tokens for the variable names. Regular Expressions are designed according to the rules specified in the Toy language definition.

Regular Definition:

Letter ⇨ [a-z, A-Z]

Digit ⇨ [0-9]

Pattern: letter(letter|digit)\*

Token : <id, path, symbol\_table>

* **Numbers**:

Digits ⇨ <digits>+

Number⇨ digits<.digits>

Token : <number, number\_value>

Example: <number, 2.5>

* **Whitespaces** :

In the lexical analysis we strip out the newline, tab and white spaces. Using the ascii value of : \n, \t and white\_space.

Pattern: ascii value of (newline, tab, spaces)\*

**Finite State Automata for Lexical Analysis :**

The main task of the lexical analyzer in a compiler is to scan the input text and produce a stream of tokens for the parser to perform syntax analysis. Finite State Automata helps us in recognizing the patterns. It acts as a directed graph with set of notes representing individual states.

We can construct a State Transition Diagram by alphabets:

 Set of States:

F

2aaaaaaaa

2

1

**aaaaaaaaa**

0

 **a a b**

Implementation:

We are given a string x: aabbb

If we are in the final state at the end then machine will accept string x.

L(M): Set of accepted strings by machine M.

**Example**: FSM for identifying ‘while’ keyword

 w h i l e **digit**/

F

hhhhhhhh2aaaaaaaa

**waaaaaaaaa**

0

1

2

5

4

3

 **letter**

Backtrack

Each state represents the condition we have between Lexeme Begin pointer and Lexeme Forward pointer.

**Lexeme Begin**: The begin pointers always points at the beginning of the lexeme to be

recognized.

**Lexeme Forward**: The forward pointer scans ahead the input until there are no more next

states in the Automaton—we are sure that the longest lexeme has been

found.

We construct token for each keyword, i.e., state transition diagram for each token.

**State Transition Machine for Operators**:

2aaaaaaaa

F

1

**<<aaaaaaaaa**

0

 **< =**

F

2aaaaaaaa

 **other** **character**

Backtrack

**State Transition Diagram for Identifiers:**
 **digit/letter**

0

**aaaaaaaaa**

1

F

 **letter**  **not letter/**

2aaaaaaaa

 **digit**

* How to distinguish between keywords and identifiers?

Approach1:

 Mark keywords in Symbol Table (ST) with their names and corresponding tokens.

if(lexeme==keyword in ST)

 {

 return token;

}

else

{

 insert new token in symbol table;

}

Approach 2:

Constructing State Transition diagram for both keywords and variables. First we check if a pattern is keyword, otherwise if it is a digit or a letter than we recognize it as a variable after recognizing it in the variable\_name FSM. If it is not recognized by the variable\_name FSM, then we output an error.