Compiler Theory

(Lexical Analysis)

003

The Role of a Lexical Analyzer

- The main tasks of the LA are to :-
  - read the input characters of the source program;
  - group them into lexemes (words in the language), and
  - produce as output a sequence of tokens for each lexeme in the source program.

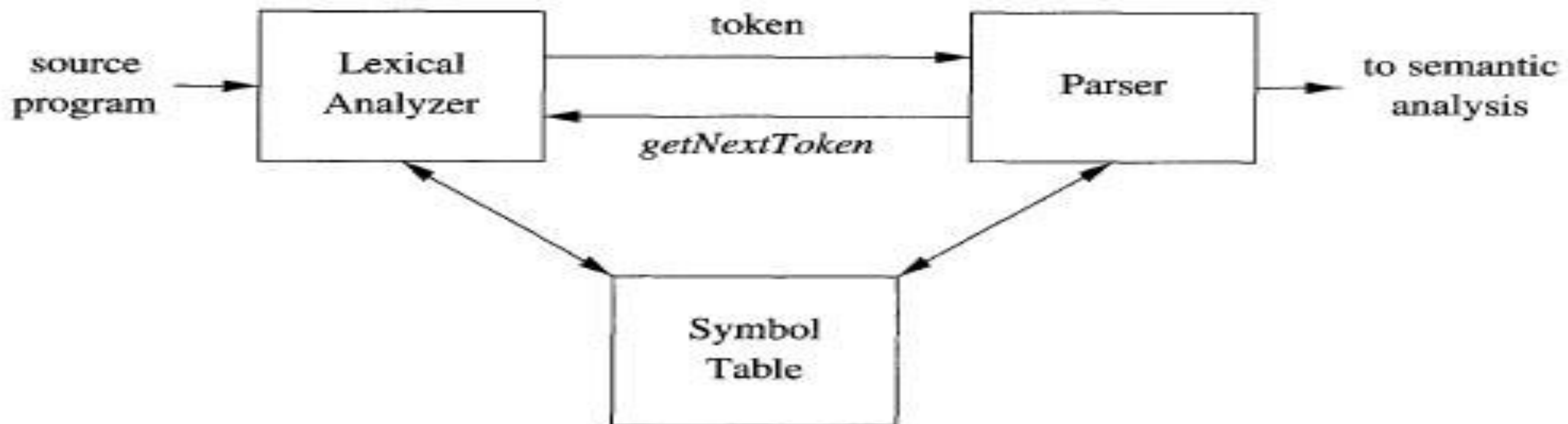


Figure 3.1: Interactions between the lexical analyzer and the parser

Implementation of a Lexical Analyzer

- There are really two options;
- We can either create a lexical analyzer from scratch … that is implement a program to identify lexemes; or else
- (what we shall be doing) … produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical-analyzer generator and compiling those patterns into code that functions as a lexical analyzer.
- We shall be using (J)(F)Lex and JavaCC

Tokens, Patterns and Lexemes

- A token is a pair <name, attribute>. The name is an abstract symbol representing a kind of lexical unit, e.g. a keyword, identifier. The attribute may store things such as it’s lexeme, type, location at which it is first found (for possible error reporting at later stages)

- A pattern is a description of the form that the lexemes of a token may take. e.g. regular expression

- A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token

Set of tokens …

- Keywords :- One token for each (if, while, etc.)
- Operators :- tokens for the operators (e.g. comparison token <, <= ,==)
- Id :- One token representing all identifiers
- Constants :- e.g. number and literal tokens
- Punctuation :- token for each punctuation symbol (e.g. left parenthesis, comma, etc)

Microsyntax

- The microsyntax of a language describes how elements of the alphabet are grouped together, e.g. Alphabetic letters are grouped from left to right to form a word and a blank space terminates a word.
- For a programming language, the scanner (lexical analyser) applies a set of rules, sometimes called the microsyntax of the language, in order to produce the tokens.
- The compiler writer starts from a specification of the microsyntax of the language ... Make sure you understand the difference between microsyntax and syntax of a language because you'll be using both for the assignment!!

Specification of Lexeme Patterns

- Regular expressions are usually (practically always) used to specify lexeme patterns (the microsyntax of the language)

- Let L = {A,B,C,…,Z,a,b,c,…,z} and let D = {0,1,…,9}
  - L U D is the set of letters and digits – the language with 62 string of length one, each of which strings is one letter or number long,
  - LD is the set of 520 strings of length two, each consisting of one letter followed by one digit,
  - L^4 is the set of all 4-letter strings,
  - L* (Kleene Closure) is the set of all strings of letters, including the empty string,
  - L(L U D)* is the set of all strings and digits beginning with a letter,
  - L^+ (Positive Closure) is the set of all strings of one or more digits.

More examples of Regular Expressions

- Let Σ = {a, b}
  - a|b denotes the language {a,b}
  - (a|b)(a|b) denotes the language {aa,ab,ba,bb}
  - a* denotes the language {e,a,aa,aaa,…}
  - (a|b)* denotes the language {e,a,b,aa,ab,ba,bb,aaa,…}
  - a|a*b denotes the language {a,b,ab,aab,aaab} i.e. the string a and all strings consisting of zero or more a’s and ending in b.
- | is read or … a | b means a or b

Regular Expression for C identifiers

- Letter_ -> A | B | C | … | Z | a | b | … | _
- Digit -> 0 | 1 | … | 9
- Id -> letter_ (letter_ | digit)*

- Regular Definitions
  - Used for notational convenience to give names to certain regular expressions and use those in subsequent expressions, as if the names were themselves symbols.
  - e.g. Letter and digit above

some LEX regular expressions

- DIGIT [0-9]
- ID [a-z][a-z0-9]*
- “s”
  - String s literally
- (DIGIT)$^+$ ”.” (DIGIT)*
- r* , r$^+$
- r?
  - Zero or one r
- r{m,n}
  - Between m and n occurences of r

JavaCC Lexical Analyzer - Tokens

- "if"
- Digit = ["0"-"9"]
- Id = ["a"-"z"] (["a"-"z"]|<Digit>)*
- Num = (<Digit>)$^+$
- Real = ((<Digit>)$^+$ "." (<Digit>)*) |
  ((<Digit>)* "." (<Digit>)$^+$)>

JavaCC TOKEN and SKIP

- TOKEN is used to specify that the matched string should be transformed into a token that should be communicated to the parser.
- SKIP is used to specify that the matched string should be thrown away (e.g. comments)

- Note that the definition of DIGIT is there to be used in the definition of other tokens. In JavaCC the definition of DIGIT is preceeded by # to indicate this.

Input Buffering

- Using one system read command to read just one character from the input does not make sense .... therefore with one system read command N characters are read.
- A simple scheme uses two buffers
- Two pointers to the input are maintained
  - LexemeBegin – points to start of current lexeme begin identified
  - Forward – scans ahead until a pattern match is found.

Tokens for a simple language

(check next slide for patterns of terminals)

- *stmt*
  - **if** *expr* **then** *stmt*
  - **If** *expr* **then** *stmt* **else** *stmt*
  - ε
- *expr*
  - *term* **relop** *term*
  - *term*
- *term*
  - **id**
  - **number**

Patterns for tokens

- Digit : [0-9]
- Digits : digit$^+$
- Number : digits (. digits)? (E [+-]? digits)?
- Letter : [A-Za-z]
- Id : letter (letter | digit)*
- If : if
- Then : then
- Else : else
- Relop : < | > | <= | >= | = | <>

Attribute Values for id number and relop

- For token names **if**, **then** and **else** we don't need to assign any attribute values
- For both **id** and **number** we need to keep track of the lexeme using a pointer to their entry in symbol table.
- For **relop**, we need to keep track of which relational operator we've matched, namely LT,LE,EQ,NE,GT,GE.

Transition Diagrams

- Consist of a set of states and directed labelled transitions.
- Certain states are said to be accepting or final.
- One state is designated the start or initial state.
- Next slides shows transition diagrams for **relop**. Pg 131 Aho and id/keywords

DFA and NDFA

- Deterministic Finite State Automata
- Non-Deterministic Finite State Automata
- Equivalent Expressive Power ! (in terms of languages generated /or recognised)

- DFA are used to parse tokens.

From RE to Scanners

- Steps (using methods/constructions you've learned about from your formal languages course)
  1. Construct NFA from the REs: Recall how to do this from your formal languages course i.e. First build NFAs for each character then using the constructions for alternation, concatenation and closure. Remeber that order (implied by parethesis, precedence) is important!!!
  2. NFA to DFA: Since DFA execution is much easier to implement than NFA execution, the next step converts the NFA to a DFA. Again recall what you have done in the formal languages part of this course.
  3. DFA to Minimal DFA (Hopcroft's Algorithm)

Rel*(ational)* ***Op****(erators) DFA*

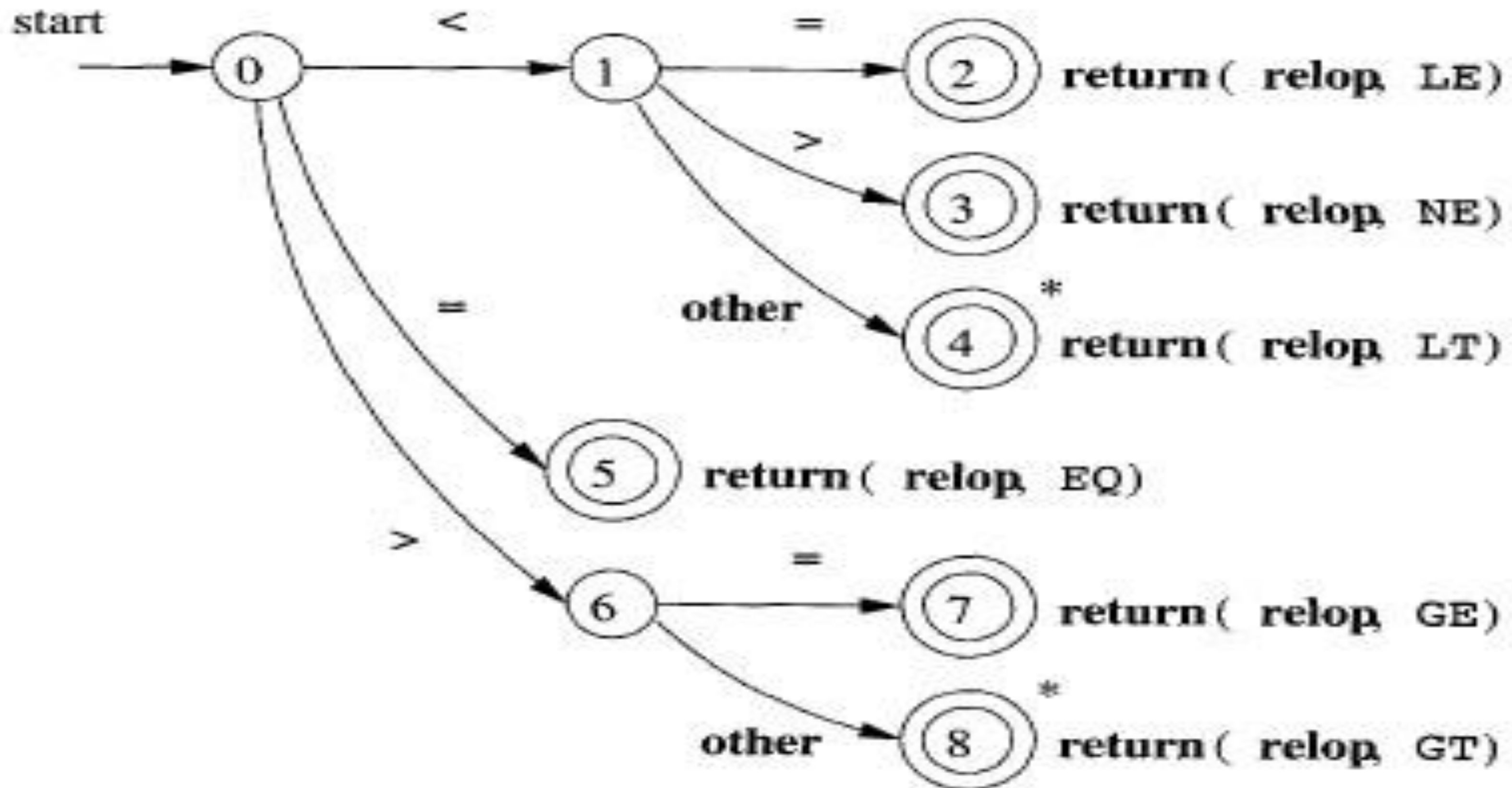


Figure 3.13: Transition diagram for **relop**

Reserved Words and Identifiers

- There are two possibilities
  - Install the reserved words in the symbol table initially ...

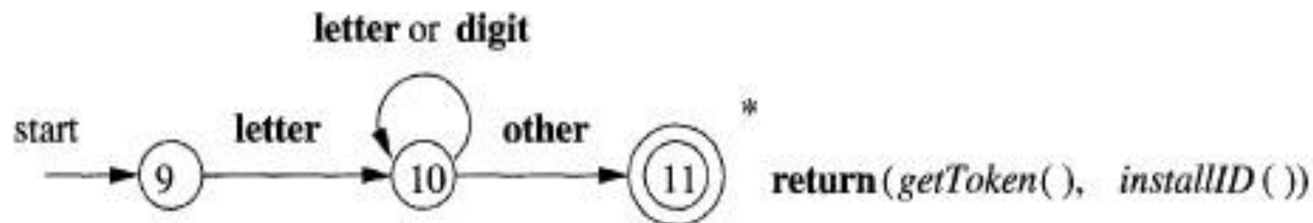


Figure 3.14: A transition diagram for **id**'s and keywords

  - Or ... create separate transition diagrams (DFA) for each keyword

DFA for delimiter token (pg 134)

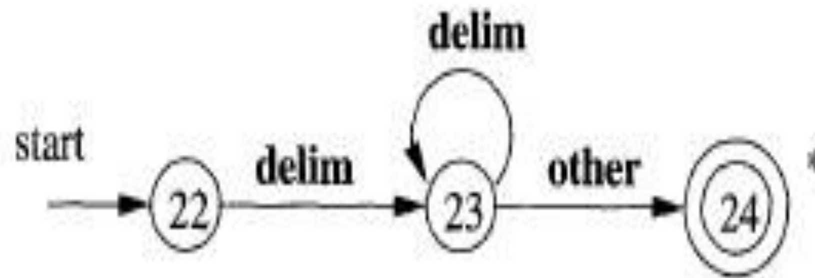


Figure 3.17: A transition diagram for whitespace

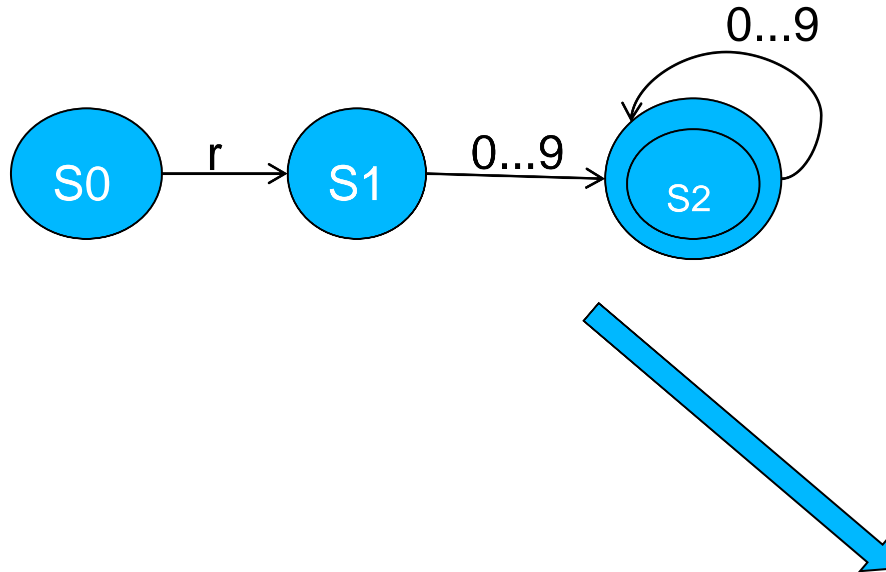
Implementing Scanners

- Once a set of REs is defined for a given language, the scanner generator must convert the DFA into executable code.
- There are three options ...
  - Table-driven scanner
  - Direct-coded scanner
  - Hand-coded scanner
- All simulate the execution of a DFA.

Table-Driven Scanner Generator

- The scanner uses a skeleton scanner for control and a set of generated tables that encode language-specific knowledge.
- Compiler writer described the lexical patterns by providing a set of regular expressions.
- The scanner then dynamically generates the required tables that drive the skeleton scanner.
- The skeleton scanner implementation is divided into four sections: *Initialisztion*, *Scanning Loop* (DFA execution), *Rollback Loop*, *Final Section* (returns token).

Scanner Tables *(eg Cooper & Torczon)*



Token Type Table

s0	s1	s2	se
invalid	invalid	register	invalid

Transition Table

	Register	Digit	Other
S0	S1	Se	Se
S1	Se	S2	Se
S2	Se	S2	Se
Se	Se	Se	Se

Classifier Table

R	0,1,…,9	EOF	Other
Register	digit	Other	Other

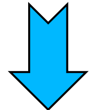
Table-Driven Scanner *(Cooper&Torczon)*

Initialisation

```
NextWord()
   state = s0;
   lexeme = " ";
   stack.clear();
   stack.push(bad);
```

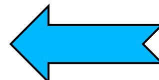
scanning loop

```
while (state != se) {
   NextChar(&char);
   lexeme = lexeme + char;
   if (state ϵ SA) stack.clear();
   push (state);
   cat = CharCat(char);
   state = δ [state, cat];
}
```



roll back loop

```
while (state != sA && state != bad) {
   state = stack.pop();
   truncate lexeme;
   Rollback();
}
```



report result

```
if (state ϵ sA )
   return Type[state];

else return invalid;
```

Rollback factor !!!

- Scanner match maximal length lexeme.
- Rollback might become a problem.
- Consider the language ((ab)|((ab)*c))* and input string abababababababababab ....
- Worst case rollback consumes quadratic time.
- We can modify the scanner in the previous slide in order to avoid this problem ....

The Maximal Munch Scanner !!

- Changes inlcude :

1. a global counter, *InputPos*, to record position in the input stream.
2. A bit array, *Failed*, to record dead end transition as the scanner finds them.
3. Set *InputPos* to zero and array *Failed* to false values in the initialisation call.

Direct-coded Scanner Generator

- Replaces tables with code represeting implicitly the state transition diagram.
- I.e. A direct coded scanner has specialised code fragments to implement each state in the DFA.
- Control is therefore transferred directly from state-fragment to state-fragment to emulate the actions of the DFA.
- Special states S_{init} (called to for each nextWord()) and S_{out} (called to possibly rollback and return a token)

Direct-coded scanner ...

- S_{init} : lexeme = "";
  clear stack;
  push(bad);
  goto S_0;

Note that not all state program fragments are listed here .. Eg s1 is not which can be reached from s0

- S_0 : NextChar(char);
  lexeme += char;
  if (state in S_A) clear stack;
  push(state);
  if (char = 'r') then goto S_1 else goto S_{out};

- S_{out} : while (state notin S_A and state neq bad)
  { state = pop(); truncate lexeme; }
  if (state in S_A) then return Type[state] else return error;

Conflict Resolution (the Lex way)

- If several prefixes of the input match one or more patterns :
  - Always prefer the longer to a shorter prefix
  - If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

The Lexical-Analyser Generator Lex

- Flex and Jflex are more recent developments on the original Lex
- Behind the scene Lex transforms the input patterns (regular expressions) into a transition diagram and generates code that simulates this transition diagram.
- lex.l -> Lex compiler -> lex.yy.c
- lex.yy.c -> C compiler -> a.out
- Input stream -> a.out -> sequence of tokens

JFlex

- jflex <options> <inputfiles>
- Then you run JFlex with:
- java JFlex.Main <options><inputfiles>

Structure of a *Lex* Program

- Declarations

  %%

  Translation rules

  %%

  Auxiliary functions

- Declarations contain variables, constants (e.g. LT, LE, EQ, etc) and regular definitions
- Translation rules each have the form Pattern { Action }
- Auxiliary functions include any functions which have been used in the actions of the transitions rules.

Jflex – Lexer States (i)

- In addition to regular expression matching lexers usually use lexical states to refine a specification.
- A lexical state acts like a start condition.
- A start condition of a regular expression can contain more than one lexical state.
- It is then matched when the lexer is in any of these lexical states.

Jflex – Lexer States (ii)

- In Lex, the lexical state YYINITIAL is predefined and is also the state in which the lexer begins scanning.
- If a regular expression has no start conditions it is matched in all lexical states.
- <STRING> {
  - expr1 { action1 }
  - expr2 { action2 }

  }
- ... means that both expr1 and expr2 have start condition <STRING>

e.g. of usage of lexical state

- “ in the input indicates we've come across a string literal.
- First clear string buffer ...
- ... then change lexical state to <STRING>
- \" { string.setLength(0); yybegin(STRING); }

- Please go through all manual pages in JFLex in order to understand the syntax for the regular expressions used in JFlex.

How does Jflex parse it's input?

- When consuming its input, the scanner determines the regular expression that matches the longest portion of the input (longest match rule).
- If there is more than one regular expression that matches the longest portion of input (i.e. they all match the same input), the generated scanner chooses the expression that appears first in the specification.
- After determining the active regular expression, the associated action is executed.
- If there is no matching regular expression, the scanner terminates the program with an error message

How does Jflex parse it's input?

- ```
  %states A, B
  %xstates C
  %%
  expr1                 { yybegin(A); action }
  <YYINITIAL, A> expr2   { action }
  <A> {
    expr3               { action }
    <B,C> expr4           { action }
  }
  ```
- The rule with expr1 has no states listed, and is thus matched in all states but the exclusive ones, i.e. A, B, and YYINITIAL.
  ```

Jflex … more details

- Are available at
  - www.jflex.de

- Part of your assignment is to understand well (more than what we've covered today) how this popular lexer works.
- Lexers are used in many other areas of CS and AI, so you might as well figure out how it works in order to save yourselves time later on.

JavaCC Lexer

- TOKEN : {

  < IF: “if” >

  < #DIGIT” [“0”-”9”] >

  < ID: [“a”-”z”] ([“a”-”z”]|<DIGIT>)* >

  <NUM: (<DIGIT>)$^+$ >

  }
- SKIP : {

  <”--” ([“a”-”z”])* (“\n” | “\r” | “\r\n”)>

  “ “

  “\t”

  “\n”

  }