Lexical Analysis

The role of a lexical analyser

- The first phase of a compiler
- Reads input characters
- Groups them into lexemes
- Produces as output a sequence of tokens
- The stream of tokens is sent to the parser
- When it discovers the lexeme for a new identifier it enters this lexeme into the symbol table

0.1 Lexical Analysis vs Parsing

Lexical analysis and syntax analysis are separate phases

- Simplicity of design
  - Simplify each task separately
- Improved compiler efficiency
  - Apply specialised techniques for each step
- Compiler portability
  - Different lexical analysers for specific devices

1 Tokens vs Lexemes

Token

- A notation representing the kind of lexical unit, e.g.
  - A keyword
  - An identifier
- Consists of a pair of:
  - A token name
  - An (optional) attribute value
- The tokens are given as input to the parser

Pattern of a token

- A description of the form of its lexemes, e.g.
  - For a keyword, the sequence of its characters
  - For an identifier: a description that matches many strings

The lexeme of a token

- A sequence of characters in the source program that matches the pattern of the token
- The lexical analyser identifies a lexeme as an instance of a token

2 Attributes of tokens

In cases when many lexemes match with a specific token

- The compiler must know which lexeme was found in the source program
- The lexical analyser provides to the parser
  - The token name
  - Additional information that specifies the particular lexeme represented by the token (attribute value)

Token names - influence parsing decisions

Attribute names - influence the transition of the token after the parsing (in the semantic analysis)

Identifiers(token name is id):

- Its lexeme (i.e. the sequence of its characters)
- Its type (e.g. integer, float, boolean)
- The line of the source program it has been found(in order to report error messages)
- etc

All this information is stored in the symbol table

3 Recognition of tokens

The source program can have several keywords:

- e.g. if, then, else, for
- They are reserved words, i.e. can not be used as identifiers
- Their tokens have no attribute value

Patterns of tokens can be:

- Expressed by regular expressions
- Defined by regular definitions (i.e. no recursive definitions)

Syntax of a program can be expressed by a context free grammar

A simple context free grammar for branching statements:

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term} \textbf{ relop } \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Terminal symbols of the grammar:

- The token names, e.g. if, then, else, relop, id, number
- number: a ”constant”; relop: for a ”comparison operator”

Regular definitions for these operators

$$
\begin{array}{rcl}
\textit{digit} & \rightarrow & \texttt{[0-9]} \\
\textit{digits} & \rightarrow & \textit{digit}^{+} \\
\textit{number} & \rightarrow & \textit{digits}\ (\texttt{.}\ \textit{digits})? \\
\textit{letter} & \rightarrow & \texttt{[A-Za-z]} \\
\textit{id} & \rightarrow & \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^{*} \\
\textit{if} & \rightarrow & \texttt{if} \\
\textit{then} & \rightarrow & \texttt{then} \\
\textit{else} & \rightarrow & \texttt{else} \\
\textit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Important notes:

- A superscript + means one or more
- A superscript * means zero or more
- A question mark at the end of brackets means optional

There is also an additional rule for stripping out whitespace

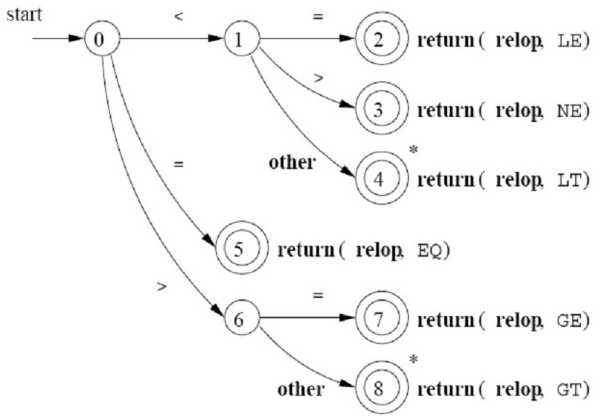
$$ws \rightarrow (\text{ blank} \mid \text{tab} \mid \text{newline })$$

All tokens with their attribute value and their corresponding lexemes

Lexemes	Token Name	Attribute Value
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

4 Transition diagrams

- When we scan the source diagram to identify lexemes, we need two pointers
  - `lexemeBegin`: marks the beginning of the current lexeme
  - `forward`: scans ahead until a pattern match is found
- Patterns for tokens can be
  - Expressed by regular expressions, and thus
  - Recognised by a DFA
- We model the recognition of a pattern by a transition diagram (a special type of a DFA)
  - States represent all the current information between `lexemeBegin` and `forward`
  - Actions are attached to the final states (to be performed after the lexeme has been scanned)
- Types of actions attached to the final states:
  - Return a specific token name and attribute value
  - Retract the forward pointer one position (indicated by an *)
  - Just proceed to the next character (for empty spaces),etc
  - This is an example of a transition diagram for the token relop



5 Keywords vs Identifiers

- Keywords like for, if then else are reserved words and so can't be declared as identifiers even though they look like identifiers
- Usually we look for identifier lexemes with a transition diagram

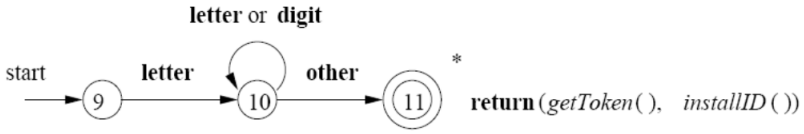


Figure 3.14: A transition diagram for **id**'s and keywords

Two ways to handle reserved words:

1. Install the reserved words initially in the symbol table, then:
   - The corresponding entry in the symbol table stored information that this is a reserved word
   - When we scan a lexeme, a call to installID
     - Checks in the symbol table whether this lexeme is a reserved word
     - It adds it in the symbol table (as an id) only if its not already there
2. Create separate transition diagrams for each reserved word, example for then

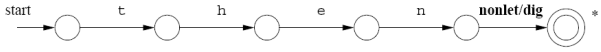


Figure 3.15: Hypothetical transition diagram for the keyword `then`

   - We must check that the scanned word ends after reading then, otherwise it might be an identifier containing the word then
   - For each new lexeme:
     - First run the transition diagrams for reserved words
     - If no accepts, the token can be recognised as id

6 Pattern Matching

There exist tools to automatically generate a lexical analyser just by specifying the regular expressions to describe patterns to tokens

An example of this is Lex, it performs as follows:

- Lists some patterns (regular expressions) with some order

- Scans the input
  - Until it finds the longest prefix of the input that matches one of the patterns
  - If it matches many patterns, it chooses the first one in the order
- Returns the corresponding token to the parser

Suppose Lex has three patterns

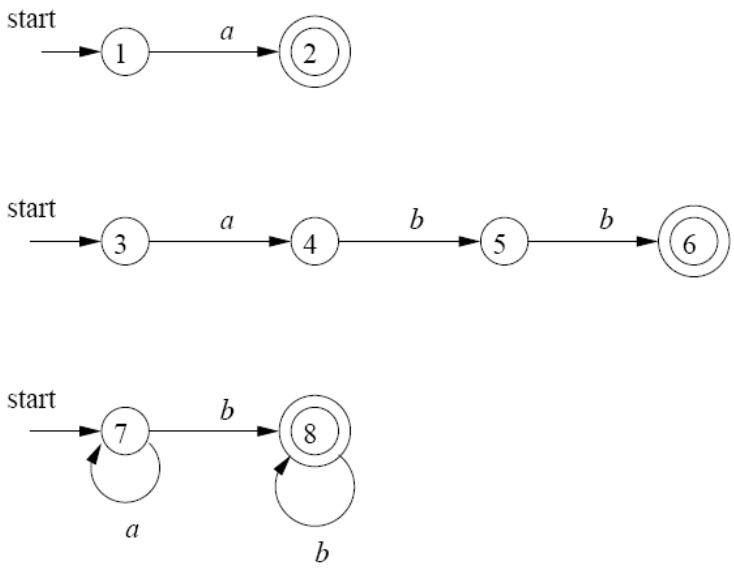


Figure 3.51: NFA's for **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

It composes a single equivalent NFA that recognises all of them

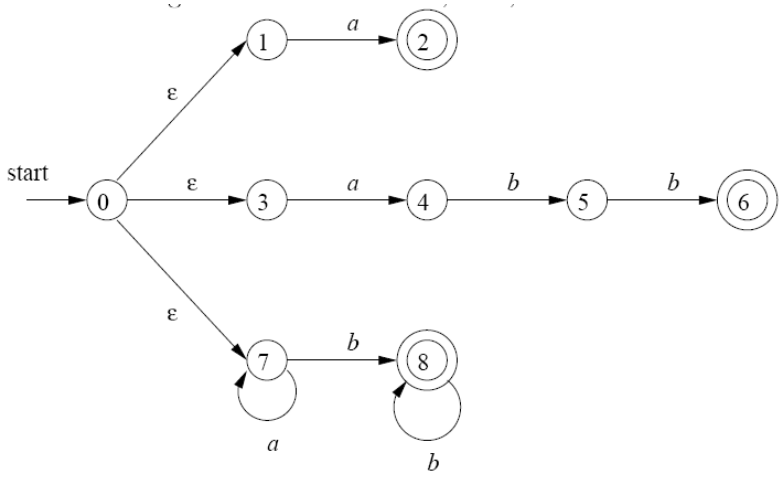


Figure 3.52: Combined NFA

As it moves ahead:

- It calculates the set of states it is at each point
- It simulates the NFA until there are no next states
- After that:
  - There can not be any longer prefix
  - It looks backward, until it finds a set of states that includes a final state
  - It picks the accepting state associated with the earliest pattern
  - It returns this pattern