Lexical Analysis
The Scanner
CSC 4181 Compiler Construction

Introduction

• A scanner, sometimes called a lexical analyzer
• A scanner:
  – gets a stream of characters (source program)
  – divides it into tokens
  • Tokens are units that are meaningful in the source language.
  • Lexemes are strings which match the patterns of tokens.

Examples of Tokens in C

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>Age, grade, Temp, zone, q1</td>
</tr>
<tr>
<td>number</td>
<td>3.1416, -498127, 987.76412097</td>
</tr>
<tr>
<td>string</td>
<td>“A cat sat on a mat.”, “90183654”</td>
</tr>
<tr>
<td>open parentheses</td>
<td>(</td>
</tr>
<tr>
<td>close parentheses</td>
<td>)</td>
</tr>
<tr>
<td>Semicolon</td>
<td>;</td>
</tr>
<tr>
<td>reserved word if</td>
<td>IF, if, If, iF</td>
</tr>
</tbody>
</table>

Scanning

• When a token is found:
  – It is passed to the next phase of compiler.
  – Sometimes values associated with the token, called attributes, need to be calculated.
  – Some tokens, together with their attributes, must be stored in the symbol/literal table.
  • It is necessary to check if the token is already in the table
• Examples of attributes
  – Attributes of a variable are name, address, type, etc.
  – An attribute of a numeric constant is its value.

How to construct a scanner

• Define tokens in the source language.
• Describe the patterns allowed for tokens.
• Write regular expressions describing the patterns.
• Construct an FA for each pattern.
• Combine all FA’s which results in an NFA.
• Convert NFA into DFA
• Write a program simulating the DFA.

Regular Expression

• λ  a character or symbol in the alphabet
• φ  an empty string
•  an empty set
• if r and s are regular expressions
  • r | s
  • rs
  • r*
  • (r)
Extension of regular expr.

- [a-z] — any character in a range from a to z
- . — any character
- r+ — one or more repetition
- r? — optional subexpression
- ~(a | b | c),[^abc] — any single character NOT in the set

Examples of Patterns

- (a | A) = the set {a, A}
- [0-9]* = (0 | 1 | ... | 9) (0|1 | ... | 9)*
- [0-9]? = (0 | 1 | ... | 9 |
- [A-Za-z] = (A | B | ... | Z | a | b | ... | z)
- A . = the string with A following by any one symbol
- ^0-9] = [^0123456789] = any character which is not 0, 1, ..., 9

Describing Patterns of Tokens

- reservedIF = (IF | if | If | iF) = (I|i)(F|f)
- letter = [a-zA-Z]
- digit = [0-9]
- identifier = letter (letter|digit)*
- numeric = (+|-)? digit+ (. digit*)? (E (+|-)? digit*)?
- Comments
  - { (~})* // from tiny C grammar
  - /* ([^*]*[^*/]*) */ // C-style comments
  - ;(~newline)* newline // Assembly lang comments

Disambiguating Rules

- IF is an identifier or a reserved word?
  - A reserved word cannot be used as identifier.
  - A keyword can also be identifier.
- <= is < and = or <=?
  - Principle of longest substring
  - When a string can be either a single token or a sequence of tokens, single-token interpretation is preferred.

Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) is a mathematical model that consists of
1. A set of states S
2. A set of input symbols \(\Sigma\)
3. A transition function that maps state/symbol pairs to a set of states: \(S \times (\Sigma + \epsilon) \rightarrow \text{set of } S\)
4. A special state \(s_0\) called the start state
5. A set of states \(F\) (subset of \(S\)) of final states

INPUT: string
OUTPUT: yes or no

Example NFA

Transition Table:

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>(\epsilon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\(S = \{0,1,2,3\}\)
\(S_0 = 0\)
\(\Sigma = \{a,b\}\)
\(F = \{3\}\)
NFA Execution

An NFA says ‘yes’ for an input string if there is some path from the start state to some final state where all input has been processed.

NFA(int s0, int input_element) {
  if (all input processed and s0 is a final state) return Yes;
  if (all input processed and s0 is not a final state) return No;
  for all states s1 where transition(s0, table[input_element]) = s1
    if (NFA(s1, input_element + 1) = Yes) return Yes;
  for all states s1 where transition(s0, e) = s1
    if (NFA(s1, input_element) = Yes) return Yes;
  return No;
}

Uses backtracking to search all possible paths.

Deterministic Finite Automata

A deterministic finite automaton (DFA) is a mathematical model that consists of:
1. A set of states $S$
2. A set of input symbols $\Sigma$
3. A transition function that maps state/symbol pairs to a state: $S \times \Sigma \rightarrow S$
4. A special state $s_0$ called the start state
5. A set of states $F$ (subset of $S$) of final states

INPUT: string
OUTPUT: yes or no

FA Recognizing Tokens

- Identifier
- Numeric
- Comment

Example

- identifier = letter(letter|digit)*

Combining FA's

- Identifiers
- Reserved words
- Combined

Lookahead

- letter, digit Return ID
- letter, digit Return IF
Implementing DFA

- nested-if
- transition table

Nested IF

switch (state)
{
case 0:
    if isletter(nxt)
        state=1;
    elseif isdigit(nxt)
        state=2;
    else state=3;
    break;
}
case 1:
    if isletVdig(nxt)
        state=1;
    else state=4;
    break;
}
...

Transition table

<table>
<thead>
<tr>
<th>St</th>
<th>ch</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>letter</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>digit</td>
<td>2</td>
<td>1</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram of DFA

Diagram of Nested IF