CS3300 - Compiler Design

Lexical Analysis

KC Sivaramakrishnan

IIT Madras

Lexical analysis

- Also known as scanning.
- Reads a stream of characters and groups them into meaningful sequences, called <u>lexemes</u>.
- Eliminates white space

Lexical analysis - Example

Consider the following program snippet:

```
if (i==j)
  z=0;
else
  z=1;
```

- The program is just a string of characters:
  - `\tif (i==j)\n\t\tz=0;\n\n\telse\n\t\tz=1;`
- The goal of lexical analysis is to take the above string as input and partition it into substrings.
- Each substring will correspond to a token.

What is a Token?

- A syntactic category
  - In English: noun, verb, adjective, ...
  - In a programming language: Identifier, Constant, Keyword, Whitespace, ...
- Each token corresponds to a set of strings, which is described using a pattern.
- A string which matches the pattern of a token is called a lexeme.
- Lexical Analyzer produces a stream of tokens, along with relevant attribute-values for each token.
  - This stream is then sent as input to the parser.

Tokens

- For each lexeme, the lexical analyzer produces an output of the form:
  ⟨token-type, attribute-values⟩
- Example token-types: identifier, number, string, operator and …
- Example attribute-types: token index, token-value, line and column number and …
- Example:
  - `position = initial + rate * 60`
  - For a typical language like C/Java the following lexemes and their values can be identified:

lexeme	token
`position`	⟨id, position⟩
`=`	⟨op, =⟩
`initial`	⟨id, initial⟩

lexeme	token
`+`	⟨op, +⟩
`rate`	⟨id, rate⟩
`*`	⟨op, *⟩
`60`	⟨num, 60⟩

Specifying patterns

A lexical analyzer must recognize the units of syntax

- identifiers
  an alphabet followed by any number of alphanumerics
- numbers
  - integers: 0 or digit from 1-9 followed by digits from 0-9
  - decimals: integer '.' digits from 0-9
  - reals: (integer or decimal) 'E' (+ or -) digits from 0-9

We need a powerful notation to specify these patterns

Regular Expressions

Patterns are often specified as regular languages
Notations used to describe a regular language include both regular expressions and regular grammars
Regular expressions (over an alphabet Σ):

1. ε is a RE denoting the set $\{\varepsilon\}$
2. if $a \in \Sigma$, then a is a RE denoting $\{a\}$
3. if r and s are REs, denoting $L(r)$ and $L(s)$, then:

   r is a RE denoting $L(r)$
   $r \mid s$ is a RE denoting $L(r) \bigcup L(s)$
   rs is a RE denoting $L(r)L(s)$
   r^* is a RE denoting $L(r)^*$

Exercise

```
float square(float x){
  return x*x;
}
```

Perform lexical analysis for the above function. Determine the stream of tokens that the lexical analyzer would return.

What would be an example of an error caught by the lexical analyzer?

Errors caught by lexer

- Unterminated strings – `"` without a matching `"`
- Unterminate comments – $/*$ but no matching $*/$
- Exceeding identifier or numeric constant length
- Illegal characters – $\alpha, \beta, \ldots$ and other unicode characters

Examples of Regular Expressions

- identifier
  $\underline{\text{letter}} \rightarrow (a \mid b \mid c \mid ... \mid z \mid A \mid B \mid C \mid ... \mid Z)$
  $\underline{\text{digit}} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$
  $\underline{\text{id}} \rightarrow \underline{\text{letter}}\ (\ \underline{\text{letter}} \mid \underline{\text{digit}}\ )^*$
- numbers
  $\underline{\text{integer}} \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid 3 \mid ... \mid 9)\ \underline{\text{digit}}^*)$
  $\underline{\text{decimal}} \rightarrow \underline{\text{integer}}\ .\ \underline{\text{digit}}\ \underline{\text{digit}}\ ^*$
  $\underline{\text{real}} \rightarrow (\ \underline{\text{integer}} \mid \underline{\text{decimal}}\ )\ \mathtt{E}\ (+ \mid -)\ \underline{\text{digit}}\ \underline{\text{digit}}^*$

Most tokens can be described with REs
We can use REs to build lexical analyzers automatically

New Notation in REs

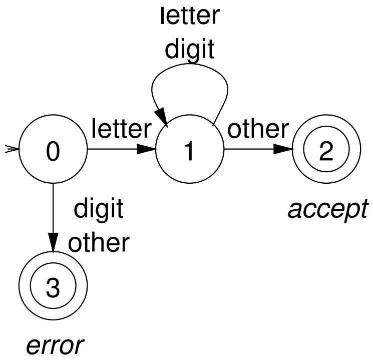
- identifier
  letter $\rightarrow [a-zA-Z]$
  digit $\rightarrow [0-9]$
  id $\rightarrow$ letter (letter | digit)*
- numbers
  integer $\rightarrow [+-]?\ (0 \mid [1-9]$ digit$^*)$
  decimal $\rightarrow$ integer . digit $^+$
  real $\rightarrow$ (integer | decimal) E $[+-]$ digit$^+$

$[a_1 a_2 \dots a_n]$ for $a_1 \mid a_2 \mid \dots \mid a_n$
r^+ for rr^* (one or more occurences)
$r?$ for $r \mid \varepsilon$ (zero or one occurence)

Recognizers

From a regular expression we can construct a
deterministic finite automaton (DFA)

Recognizer for identifier:



Code for the recognizer

Given an automata, can we write a recognizer for a token?

```
ch=nextChar();
state=0; // initial state
done=false;
tokenVal=""// empty
while (not done) {
 class=charClass[ch];
 state=
   nextState[class,state];
```

Tables for the recognizer

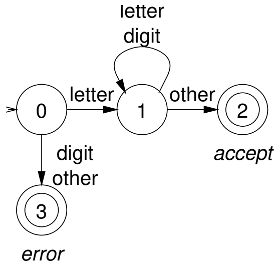
Two tables control the recognizer

`charClass:`

char	$a-z$	$A-Z$	$0-9$	other
class	letter	letter	digit	other

`nextState:`

	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—



Question: How can you implement the `charClass` table?

Code for the recognizer

Given an automata, can we write a recognizer for a token?

```
ch=nextChar();
state=0; // initial state
done=false;
tokenVal=""// empty
while (not done) {
 class=charClass[ch];
 state=
   nextState[class,state];
 switch(state) {
  case 1:
    tokenVal=tokenVal+ch;
    char=nextChar();
    break;
```

```
case 2: // accept state
      tokenType=id;
      done = true;
      break;
    case 3: // error
      tokenType=error;
      done=true;
      break;
  } // end switch
} // end while
return tokenType;
```

Tables for the recognizer

Two tables control the recognizer

`charClass:`

char	$a-z$	$A-Z$	$0-9$	other
class	letter	letter	digit	other

`nextState:`

	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

To change languages, we can just change tables

To summarize

1. Write a regular expression for each token type in the programming language. Let $R_1, \ldots, R_k$ be the regular expressions.
2. Let the input be $x_1 x_2 \ldots x_n$
   For $1 \leq i \leq n$, for $1 \leq j \leq k$, check if $x_1 x_2 \ldots x_i \in L(R_j)$
   - We can run the recognizers for all the tokens (possibly in parallel) on the input string.
3. If yes, then we have found that $x_1 \ldots x_i$ has token-type j.
4. Remove $x_1 \ldots x_i$ from input and go to step (2).

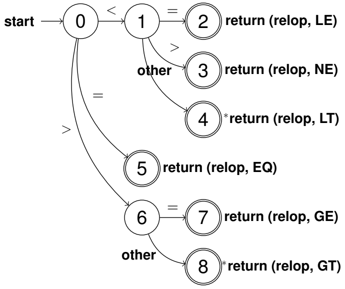
What are the issues with this algorithm?

Ambiguities in the Algorithm - I

- How much input to be used if multiple prefixes match the pattern?
  - $x_1 \ldots x_{i1} \in L(R_j)$
  - $x_1 \ldots x_{i2} \in L(R_j)$
- Examples
  - 5, 5.12, $5.12\texttt{E}{-}10$ all match the pattern for the token `digit`.
  - The string $<=$ matches the token for relational operators with value `LE` or two tokens with values `LT` and `EQ`.
- The "maximal munch" rule: Pick the longest possible prefix which matches any pattern.

Rule in action

- Transition diagram for picking the longest matching prefix for relational operations



Lookahead

- Note that in the previous example, we have to scan one symbol beyond the current lexeme to determine that the current lexeme matches the pattern.
  - Thus, for relational operators, the 'lookahead' parameter is 1.
- What is the lookahead for identifiers? For digits?
- Generally the lookahead is 1 for most tokens in modern programming languages.
- However, it could also be arbitrarily long. For example, in FORTRAN, keywords are not reserved words.
  - Hence, we could have statement such as `IF(I,J) = 3`, where `IF` is an array. But then, how to distinguish this from `IF( condition ) THEN ...`?
- We must lookahead beyond the closing parenthesis for a `THEN`.

Ambiguities in the Algorithm - II

- What happens if a prefix matches the pattern of multiple tokens?
  - $x_1 \dots x_i \in L(R_{j1})$
  - $x_1 \dots x_i \in L(R_{j2})$
- Example
  - `then` matches the pattern for keyword then and identifier id
  - We generally assume that keywords are reserved, and hence `then` must be matched to the token for keyword then.
- Two ways to resolve this problem:
  - We can recognize the lexeme as an identifier, and then include a separate check for whether the value of the identifier matches a keyword, in which case we change the token to the keyword.
  - We recognize the lexeme as both keyword and identifier, but have a total ordering among tokens to decide the winner in case of a tie (in the above case, it would be keyword).

Error recovery

- It is hard to tell (without the aid of other components), if there is a source code error.
- For example:

  ```
  fi (a == f(x))
  ```

  If `fi` a misspelling for "`if`", or a function identifier?
- Since `fi` is a valid lexeme for the token `id`, the lexer must return the token ⟨id, fi⟩.
- A later phase (parser or semantic analyzer) may be able to catch the error.

Recovery (if the lexer is unable to proceed, that is):

- Panic and stop!
- Delete one character!
- Many other one character related fixes (for example, trying a different character in place of the input)
- In any case, it is desirable to for the lexer to not get stuck, identify the error and possibly proceed with the rest of the program.

Limits of regular languages

Not all languages are regular
One cannot construct DFAs to recognize these languages:

- $L = \{p^k q^k\}$
- $L = \{wcw^r \mid w \in \Sigma*\}$

<u>Note: neither of these is a regular expression!</u>
<u>(DFAs cannot count!)</u>
But, this is a little subtle. One can construct DFAs for:

- alternating 0's and 1's
  $(\varepsilon \mid 1)(01)*(\varepsilon \mid 0)$
- sets of pairs of 0's and 1's
  $(01 \mid 10)+$

Automatic construction

Lexical Analyzer generators automatically construct code from RE-like descriptions

- construct a DFA
- use state minimization techniques
- emit code for the lexical analyzer
  (table driven or direct code)

`lex/flex` is a lexical analyzer generator

- Takes a specification of all the patterns as a RE.
- emits C code for scanner
- provides macro definitions for each token
  (used in the parser)