Chapter 4

Lexical analysis

Concepts

- Lexical scanning
- Regular expressions
- DFAs and FSAs
- Lex

Lexical analysis in perspective

- LEXICAL ANALYZER: Transforms character stream to token stream
  - Also called scanner, lexer, linear analysis

LEXICAL ANALYZER
- Scans Input
- Removes whitespace, newlines, ...
- Identifies Tokens
- Creates Symbol Table
- Inserts Tokens into symbol table
- Generates Errors
- Sends Tokens to Parser

PARSER
- Performs Syntax Analysis
- Actions Dictated by Token Order
- Updates Symbol Table Entries
- Creates Abstract Rep. of Source
- Generates Errors

This is an overview of the standard process of turning a text file into an executable program.
Where we are

```
Total = price + tax;
```

Lexical analyzer

```
Total = price + tax ;
```

Parser

```
assignment
id = Expr
id + id
price tax
```

Basic terminologies in lexical analysis

- **Token**
  - A classification for a common set of strings
  - Examples: `<identifier>`, `<number>`, etc.

- **Pattern**
  - The rules which characterize the set of strings for a token
  - Recall file and OS wildcards (* . java)

- **Lexeme**
  - Actual sequence of characters that matches pattern and is classified by a token
  - Identifiers: x, count, name, etc…

Examples of token, lexeme and pattern

```
If (price + gst – rebate <= 10.00) gift := false
```

Examples of token, lexeme and pattern

<table>
<thead>
<tr>
<th>Token</th>
<th>lexeme</th>
<th>Informal description of pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>if</td>
<td>If</td>
</tr>
<tr>
<td>Lparen</td>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>Identifier</td>
<td>price</td>
<td>String consists of letters and numbers and starts with a letter</td>
</tr>
<tr>
<td>operator</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Identifier</td>
<td>gst</td>
<td>String consists of letters and numbers and starts with a letter</td>
</tr>
<tr>
<td>operator</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Identifier</td>
<td>rebate</td>
<td>String consists of letters and numbers and starts with a letter</td>
</tr>
<tr>
<td>Operator</td>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>constant</td>
<td>10.00</td>
<td>Any numeric constant</td>
</tr>
<tr>
<td>rparen</td>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>Identifier</td>
<td>gift</td>
<td>String consists of letters and numbers and starts with a letter</td>
</tr>
<tr>
<td>Operator</td>
<td>:=</td>
<td>Assignment symbol</td>
</tr>
<tr>
<td>Identifier</td>
<td>false</td>
<td>String consists of letters and numbers and starts with a letter</td>
</tr>
</tbody>
</table>

Regular expression

- Scanners are usually based on **regular expressions**.
- Remember language is a set of strings.
- Examples of regular expression
  - letter → a|b|c|...|z|A|B|C|...|Z
  - digit → 0|1|2|3|4|5|6|7|8|9
  - identifier → letter(letter|digit)*
- Basic operations:
  - Set union
  - Concatenation
  - Kleene closure
Formal language operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>union of L and M</td>
<td>L ∪ M</td>
<td>L ∪ M = {s</td>
<td>s is in L or s is in M}</td>
</tr>
<tr>
<td>concatenation of L and M</td>
<td>LM</td>
<td>LM = {st</td>
<td>s is in L and t is in M}</td>
</tr>
<tr>
<td>Kleene closure of L</td>
<td>L*</td>
<td>L* denotes zero or more concatenations of L</td>
<td>All the strings consists of &quot;a&quot; and &quot;b&quot;, plus the empty string. {ε, a, b, a, ab, ba, aab, ...}</td>
</tr>
<tr>
<td>positive closure of L</td>
<td>L+</td>
<td>L+ denotes one or more concatenations of &quot;L&quot;</td>
<td>All the strings consists of &quot;a&quot; and &quot;b&quot;.</td>
</tr>
</tbody>
</table>

Regular expression

- Regular expression: constructing sequences of symbols (Strings) from an alphabet.
- Let Σ be an alphabet, r a regular expression then L(r) is the language that is characterized by the rules of r
- Definition of regular expression
  - ε is a regular expression that denotes the language {ε}
  - If a is in Σ, a is a regular expression that denotes {a}
  - Let r and s be regular expressions with languages L(r) and L(s). Then
    » (r) | (s) is a regular expression  \( L(r) ∪ L(s) \)
    » (r)(s) is a regular expression \( L(r) L(s) \)
    » (r)* is a regular expression \( (L(r))^* \)
- It is an inductive definition!
- Distinction between regular language and regular expression

Regular expression example revisited

- Examples of regular expression
  - letter → a|b|c|...|z|A|B|C|...|Z
  - digit → 0|1|2|3|4|5|6|7|8|9
  - identifier → letter(letter|digit)*
- Q: why it is an regular expression?

Precedence of operators

- * is of the highest precedence;
- Concatenation comes next;
- | lowest.
- All the operators are left associative.
- Example
  - (a) | ((b)*(c)) is equivalent to a|b*c
Properties of regular expressions

We can easily determine some basic properties of the operators involved in building regular expressions.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r[s] = s[r]</td>
<td>is commutative</td>
</tr>
<tr>
<td>r[(s</td>
<td>t)] = (r[s])t</td>
</tr>
<tr>
<td>(rs)t = r(st)</td>
<td>Concatenation is associative</td>
</tr>
<tr>
<td>r[s</td>
<td>t] = rs</td>
</tr>
<tr>
<td></td>
<td>Concatenation distributes over</td>
</tr>
</tbody>
</table>

Notational shorthand of regular expression

- One or more instance
  - L+ = L L*
  - L* = L+ | ε
  - Example
    » digits → digit digit*
    » digits → digit+

- Zero or one instance
  - L? = L|ε
  - Example:
    » Optional_fraction → digits|ε
    » optional_fraction → (.digits)?

- Character classes
  - [abc] = a|b|c
  - [a-z] = a|b|c|...|z

Regular grammar and regular expression

- They are equivalent
  - Every regular expression can be expressed by regular grammar
  - Every regular grammar can be expressed by regular expression

- Example
  - An identifier must begin with a letter and can be followed by arbitrary number of letters and digits.

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Regular grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: LETTER (LETTER</td>
<td>DIGIT)*</td>
</tr>
</tbody>
</table>

Formal definition of tokens

- A set of tokens is a set of strings over an alphabet
  - {read, write, +, -, *, /, :=, 1, 2, …, 10, …, 3.45e-3, …}

- A set of tokens is a regular set that can be defined by using a regular expression

- For every regular set, there is a deterministic finite automaton (DFA) that can recognize it
  - (Aka deterministic Finite State Machine (FSM))
  - i.e. determine whether a string belongs to the set or not
  - Scanners extract tokens from source code in the same way DFAs determine membership
**Token Definition Example**

- Numeric literals in Pascal, e.g.
  1, 123, 3.1415, 10e-3, 3.14e4
- Definition of token `unsignedNum`:
  `DIG → 0|1|2|3|4|5|6|7|8|9`
  `unsignedInt → DIG DIG*`
- Definition of `unsignedNum`:
  `unsignedInt → (( . unsignedInt ) | ε)`
  `((ε (+ | – | ε) unsignedInt) | ε)`

**Notes:**
- Recursion is not allowed!
- Parentheses used to avoid ambiguity
- It’s always possible to rewrite removing epsilons

**Simple Problem**

- Write a C program which reads in a character string, consisting of a’s and b’s, one character at a time. If the string contains a double aa, then print string accepted else print string rejected.
- An abstract solution to this can be expressed as a DFA

```
#include <stdio.h> main() {   enum State {S1, S2, S3};   enum Label {A, B};   enum State currentState = S1;   int label;   int c = getchar();   while (c != EOF) {     switch(currentState) {       case S1: if (c == 'a') currentState = S2; if (c == 'b') currentState = S1; break;       case S2: if (c == 'a') currentState = S3; if (c == 'b') currentState = S1; break;       case S3: break;     }     c = getchar();     if (currentState == S3) printf("string accepted\n"); else printf("string rejected\n");   }
```

**an approach in C**

```
#include <stdio.h> main() {   enum State {S1, S2, S3};   enum State currentState = S1;   enum State table[3][2] = {{S2, S1}, {S3, S1}, {S3, S3}};   int label;   int c = getchar();   while (c != EOF) {     if (c == 'a') label = A;     if (c == 'b') label = B;     currentState = table[currentState][label];     c = getchar();     if (currentState == S3) printf("string accepted\n"); else printf("string rejected\n");   }
```

**Using a table simplifies the program**

```
#include <stdio.h> main() {   enum State {S1, S2, S3};   enum State currentState = S1;   enum State table[3][2] = {{S2, S1}, {S3, S1}, {S3, S3}};   int label;   int c = getchar();   while (c != EOF) {     if (currentState == S3) printf("string accepted\n"); else printf("string rejected\n");   }
```
Lex

- Lexical analyzer generator
  - It writes a lexical analyzer
- Assumption
  - each token matches a regular expression
- Needs
  - set of regular expressions
  - for each expression an action
- Produces
  - A C program
- Automatically handles many tricky problems
- flex is the gnu version of the venerable unix tool lex.
  - Produces highly optimized code

Scanner Generators

- E.g. lex, flex
- These programs take a table as their input and return a program (i.e. a scanner) that can extract tokens from a stream of characters
- A very useful programming utility, especially when coupled with a parser generator (e.g., yacc)
- standard in Unix

Lex example

```
DIG [0-9]
ID [a-z][a-z0-9]*

%%

{DIG}+ printf("Integer\n");
{DIG}+.*{DIG}+ printf("Float\n");
{ID} printf("Identifier\n");
[ '\t'\n]+ /* skip whitespace */

%%

main(){yylex();}
```

A Lex Program
Simplest Example

```c
%%
ECHO;
%%
main()
{
  yylex();
}
```

Strings containing aa

```c
%%
(a|b)*aa(a|b)* {printf("Accept %s\n", yytext);}

[a|b]+ {printf("Reject %s\n", yytext);}

ECHO;
main() {yylex();}
```

Rules

- Each has a rule has a pattern and an action.
- Patterns are regular expression
- Only one action is performed
  - The action corresponding to the pattern matched is performed.
  - If several patterns match the input, the one corresponding to the longest sequence is chosen.
  - Among the rules whose patterns match the same number of characters, the rule given first is preferred.

/* scanner for a toy Pascal-like language */
%
#include <math.h> /* needed for call to atof() */
%
DIG [0-9]
ID [a-z][a-z0-9]*

```c
%%
{DIG}+ printf("Integer: %s (%d)
", yytext, atoi(yytext));
{DIG}+."{DIG}* printf("Float: %s (%g)
", yytext, atof(yytext));
if|then|begin|end printf("Keyword: %s
", yytext);
{ID} printf("Identifier: %s
", yytext);
"+|"-"|"*"|"/" printf("Operator: %s
", yytext);
"{"|"}"/* skip one-line comments */
[ \t|\n|\v|\f|\r]+ /* skip whitespace */
ECHO;
main() {yylex();}
```
### Flex’s RE syntax

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>character 'x'</td>
</tr>
<tr>
<td>.</td>
<td>any character except newline</td>
</tr>
<tr>
<td><code>[xyz]</code></td>
<td>character class, in this case, matches either an 'x', a 'y', or a 'z'</td>
</tr>
<tr>
<td><code>[abj-oZ]</code></td>
<td>character class with a range in it; matches 'a', 'b', any letter from 'j' through 'o', or 'Z'</td>
</tr>
<tr>
<td><code>[^A-Z]</code></td>
<td>negated character class, i.e., any character but those in the class, e.g. any character except an uppercase letter.</td>
</tr>
<tr>
<td><code>[^A-Z\n]</code></td>
<td>any character EXCEPT an uppercase letter or a newline</td>
</tr>
<tr>
<td><code>r*</code></td>
<td>zero or more r's, where r is any regular expression</td>
</tr>
<tr>
<td><code>r+</code></td>
<td>one or more r's</td>
</tr>
<tr>
<td><code>r?</code></td>
<td>zero or one r's (i.e., an optional r)</td>
</tr>
<tr>
<td><code>{name}</code></td>
<td>expansion of the &quot;name&quot; definition (see above)</td>
</tr>
<tr>
<td><code>&quot;[xy]\&quot;foo&quot;</code></td>
<td>the literal string: <code>\&quot;foo\&quot;</code> (note escaped <code>\</code>)</td>
</tr>
<tr>
<td><code>\x</code></td>
<td>if x is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of <code>\</code>. Otherwise, a literal 'x' (e.g., escape)</td>
</tr>
<tr>
<td><code>rs</code></td>
<td>RE r followed by RE s (e.g., concatenation)</td>
</tr>
<tr>
<td>`r</td>
<td>s`</td>
</tr>
<tr>
<td><code>&lt;&lt;EOF&gt;&gt;</code></td>
<td>end-of-file</td>
</tr>
</tbody>
</table>