6

Lexical Analysis

Concepts

- Overview of syntax and semantics
- Step one: lexical analysis
  - Lexical scanning
  - Regular expressions
  - DFAs and FSMs
  - 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

Basic lexical analysis terms

- Token
  - A classification for a common set of strings
  - Examples: <identifier>, <number>, <operator>, <open paren>, 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…
  - Integers: -12, 101, 0, …
Examples of token, lexeme and pattern

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

<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></td>
</tr>
<tr>
<td>Lparen</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>Identifier price</td>
<td>String consists of letters and numbers and starts with a letter</td>
<td></td>
</tr>
<tr>
<td>operator +</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Identifier gst</td>
<td>String consists of letters and numbers and starts with a letter</td>
<td></td>
</tr>
<tr>
<td>operator -</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Identifier rebate</td>
<td>String consists of letters and numbers and starts with a letter</td>
<td></td>
</tr>
<tr>
<td>Operator &lt;=</td>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>constant 10.00</td>
<td>Any numeric constant</td>
<td></td>
</tr>
<tr>
<td>Rparen</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>Identifier gift</td>
<td>String consists of letters and numbers and starts with a letter</td>
<td></td>
</tr>
<tr>
<td>Operator :=</td>
<td>:=</td>
<td>Assignment operator</td>
</tr>
<tr>
<td>Identifier false</td>
<td>String consists of letters and numbers and starts with a letter</td>
<td></td>
</tr>
</tbody>
</table>

Regular expressions (REs)

- Scanners are based on regular expressions that define simple patterns
- Simpler and less expressive than BNF
- Examples of a 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 are (1) set union, (2) concatenation and (3) Kleene closure
- Plus: parentheses, naming patterns
- No recursion!

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?
  - Because it only uses the operations of union, concatenation and Kleene closure
- Being able to name patterns is just syntactic sugar
- Using parentheses to group things is just syntactic sugar provided we specify the precedence and associatively of the operators (i.e., `|`, `*` and “concat”)

Another common operator: +

- The `+` operator is commonly used to mean “one or more repetitions” of a pattern
- For example, `letter+` means one or more letters
- We can always do without this, e.g. `letter+` is equivalent to `letter letter*`
- So the `+` operator is just syntactic sugar
Precedence of operators
In interpreting a regular expression
• Parentheses scope sub-expressions
• * and + have the highest precedence
• Concatenation comes next
• | is lowest.
• All the operators are left associative
• Example
  – (A) | ((B)* (C)) is equivalent to A | B * C
  What strings does this generate or match?
  Either an A or any number of Bs followed by a C

Epsilon
• Sometimes we’d like a token that represents nothing
• This makes a regular expression matching more complex, but can be useful
• We use the lower case Greek letter epsilon, ε, for this special token
• Example:
  digit: 0|1|2|3|4|5|6|7|8|9|0
  sign: +|–|ε
  int: sign digit+

Notational shorthand of regular expression
• One or more instance
  – L+ = L L*
  – L* = L+ | ε
  – Examples
    » digits: digit digit*
    » digits: digit+
• Zero or one instance
  – L? = L L
  – Examples
    » Optional_fraction→ digits|ε
    » optional_fraction→ r 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 an arbitrary number of letters and digits.

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Regular grammar</th>
</tr>
</thead>
</table>
| ID: LETTER (LETTER | ID → LETTER ID REST
|  | ID REST → LETTER ID REST
|  | | DIGIT ID REST
|  | | EMPTY |

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 finite automaton (FA) 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 FAs determine membership

FSM = FA
• Finite state machine and finite automaton are different names for the same concept
• The basic concept is important and useful in almost every aspect of computer science
• The concept provides an abstract way to describe a process that
  – Has a finite set of states it can be in
  – Gets a sequence of inputs
  – Each input causes the process to go from its current state to a new state (which might be the same!)
  – If after the input ends, we are in one of a set of accepting states, the input is accepted by the FA
Example
This example shows a FA that determines whether a binary number has an odd or even number of 0’s, where S1 is an accepting state.

Deterministic finite automaton (DFA)
- In a DFA there is only one choice for a given input in every state
- There are no states with two arcs that match the same input that transition to different states

REs can be represented as DFAs
Regular expression for a simple identifier
Letter: a|b|c|...|z|A|B|C...|Z
Digit: 0|1|2|3|4|5|6|7|8|9
Identifier: letter (letter | digit)*

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*
unsignedNum →
(unsignedInt | ε)
(ε | ε unsignedInt | ε)

Note:
- Recursion restricted to leftmost or rightmost position on LHS
- Parentheses used to avoid ambiguity
- It's always possible to rewrite by removing εs (ε)

• Accepting states marked with a *
• NFAs with εs are nondeterministic
• NFAs are harder to implement, use backtracking
• Every NFA can be rewritten as a DFA (gets larger, tho)
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

```
Start state
b 1
\|   \|
\| a 2 \|
\|   \|
\| a, b an accepting state
```

The state transitions of a DFA can be encoded as a table which specifies the new state for a given current state and input

<table>
<thead>
<tr>
<th>input</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

one approach in C

```
#include <stdio.h>
main()
{
    enum State {S1, S2, S3};
    enum State currentState = S1;
    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");
}
```

using a table simplifies the program

```
#include <stdio.h>
main()
{
    enum State {S1, S2, S3};
    enum Label {A, B};
    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");
}
```

Lex

• Lexical analyzer generator
  – It reads a lexical analyzer

• Assumption
  – each token matches a regular expression

• Needs
  – set of regular expressions

• 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

```
> flex -ofoolex.c foo.l
> cc -ofoolex foolex.c -lfl
> more input
begin
if size > 10
then size * 3.1415
end
```

```
>foolex < input
Keyword: begin
Keyword: if
Identifier: size
Operator: >
Integer: 10
Keyword: then
Identifier: size
Operator: *
Operator: -
Float: 3.1415
Keyword: end
```
Examples

- The examples to follow can be accessed online.
- See /afs/umbc.edu/users/p/a/park/pub/331/lex

```
% ls -l /afs/umbc.edu/users/p/a/park/pub/331/lex
total 8
drwxr-xr-x 2 park faculty 2048 Sep 27 13:31 aa
drwxr-xr-x 2 park faculty 2048 Sep 27 13:32 defs
drwxr-xr-x 2 park faculty 2048 Sep 27 11:35 footranscanner
```

A Lex Program

```
... definitions ...
"DIG"{0-9}
"ID"{a-z}[a-z0-9]*
... rules ...
"DIG"{0-9}+ printf("Integer: ", yytext);
"DIG"{0-9}+."{DIG}"* printf("Float: ", yytext);
"ID" printf("Identifier: ", yytext);
"[\s\t]+ /* skip whitespace */
"-" printf("Huh?");
... subroutines ...
main() {yylex();}
```

Simplest Example

```
%%
./\nECHO;
%%
main()
{
  yylex();
}
```

- No definitions
- One rule
- Minimal wrapper
- Echoes input

Strings containing aa

```
%%
(a|b)*aa(a|b)* {printf("Accept %s\n", yytext);}
[a|b]+ {printf("Reject %s\n", yytext);}
./\nECHO;
%%
main() {yylex();}
```

Rules

- Each rule has a pattern and an action
- Patterns are regular expressions
- 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

Definitions

- The definitions block allows you to name a RE
- If the name appears in curly braces in a rule, the RE will be substituted

```
DIG [0-9]
%%
(DIG)+ printf("int: ",$s\n", yytext);
(DIG)+."{DIG}"* printf("float: ",$s\n", yytext);
- /* skip anything else */
%%
main() {yylex();}
```
/* scanner for a toy Pascal-like language */

#include <math.h> /* needed for call to atof() */

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

%%
{DIG}+ printf("Integer: %s (%d) \n", yytext, atoi(yytext));
{DIG}+"."{DIG}* printf("Float: %s (%g) \n", yytext, atof(yytext));
{if|then|begin|end} printf("Keyword: %s \n",yytext);
{ID} printf("Identifier: %s \n",yytext);
{+|-|*|/} printf("Operator: %s \n",yytext);
{[|\]^} print one-line comments */
{ [ ]+ } /* skip whitespace */
. printf("Unrecognized: %s\n",yytext);
%
main(){yylex();}

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