Flex, Flex, Flex!

CSE423 Lab 7

Overview

- Flex!
- Regular Expressions
- Start Conditions in Flex
- This week’s lab

How is the AST built?

- Something reads the input .c file char by char and spits out words based on predefined rules: **SCANNER**
  - Keywords: if, else, for, while, switch, …
  - Operators: +, -, =, <, >, ==, …
  - Everything else: ( ) { } [ ] ; # …
- Something else builds sentences with them and creates the AST based on the C grammar: **PARSER**

FLEX, the scanner

- Flex uses Regular Expressions to turn text into tokens
- Example scanner that replaces every occurrence of the string “username” with user’s username.

```
%%
username printf("%s", getlogin());
```

Flex

- Input: a .lex file specifying the behavior of the scanner
- Output: a .c(c) file that turns into a scanner when compiled

Flex, by example

```
int num_lines = 0, num_chars = 0;
%%
\n {++num_lines; ++num_chars; }
. {++num_chars; }
%%
main() {
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
    num_lines, num_chars );
}
```

FLEX

- Flex uses Regular Expressions to turn text into tokens
- Creates a function: int yylex(void)
  - Every call to yylex continues scanning from where it last left off
- Certain tokens need more information attached to them than just type
  - CONSTANT is nice, but doesn’t tell you enough
  - “Semantic Value” is associated with certain token types
  - Stored in “yylval” which is a union defined as YYSTYPE

Flex Sections

- Flex has 3 sections – separated by “%%”
  - Definitions – States, Options, “defines”
  - Rules – the regexes and actions
  - User Code – included “as is” in C code
- Anything indented or surrounded with %{ and %} will also be included “as is” in the C code
  - The %{, %} will be removed

Flex Definitions

- Names – like using #defines
  - name definition
  - Example: DIGIT   [0-9]
  - Use: {DIGIT}+”.”{DIGIT}*
    - Matches input like 3.1415926 or 2.
- Start Conditions – more later
- Options – use %option *option*
  - yylineno – keeps track of line numbers
  - And others to specify filenames, scanner behavior, if debugging enabled or not …

Flex Preamble

- C(++) code that is executed before the scanning begins
- Including header files (like tokens.h) which the scanner needs to create meaningful tokens
- Example:

```
%{
        #include <stdio.h>
        extern "C" {int yywrap(){return(1);}}
        std::vector< int > ParenCounter;
%}
```

Flex Rules

- REGEX {actions} ‘\n’
  - REGEX – the regular expression determines what the rule can match
  - {actions} – specifies what is to be done when the rule is matched
  - All rules separated by newlines

Flex Rules

- Rules are greedy!

- Does not stop at the first rule that matches the current buffer
- The rule that matches the most text wins
- If two rules both match the same amount of text, the first listed rule wins

Some Flex Functions and Variables

- yytext – contains the text matched by the rule
  - yytext gets deleted, so don't save any pointers to it
  - If you want to save yytext, you must make a copy!
- yylval – contains the semantic value of the matched token
  - yylval is a union of all possible semantic value types, defined by YOU
  - Value and type it stores depends on token type
- yylineno – current line

Regular Expressions

- Regular expressions are instructions that tell the computer how to match text
- Looks like:
  - [a-zA-Z0-9_\.-]+\@[a-zA-Z0-9_-]+\.[\.a-zA-Z]+
- Like telling you – "Scan until you find"
  - A word (can have underscore, period)
  - Followed by an @
  - Followed by a word
  - Followed by a period
  - Followed by other words that can be separated by periods

Regular Expressions

- Let’s break that RE down
  - [a-zA-Z0-9_\.-]
  - [] mean a class
    - A class matches anything inside the brackets
    - So [aeiou] matches one vowel
  - a-z means a range
    - Matches anything that occurs between ‘a’ and ‘z’
  - _ matches an _
  - \. matches a period
    - Why ‘\.’ and not ‘.’? ‘.’ is a special regular expression character

Regular Expressions

- So we understand the word, what about the rest?
- +
  - This matches one or more
  - So [robin]+ matches ‘rrr’ ‘rob’ ‘rbn’ ‘rnbiorb’ but does not match ‘roba’ ‘goober’ or even ‘Robin’
- \@
  - Why \@? @ isn’t a special RE character?!
  - If you escape a non-special character, the regular expression ignores the \!
  - That means \@ = @

Regular Expressions

- Other important characters
  - ‘.’ – means match any character, but only one character so ‘.at’ matches ‘cat’ ‘bat’ ‘hat’ but not ‘flat’
  - ‘*’ – means zero or more so ‘r*obin’ matches ‘robin’ and ‘obin’ and ‘rrrrrrobin’
  - ‘?’ – means match zero or one so ‘ro?bin’ matches ‘robin’ ‘rbin’ but not ‘roooobin’
  - “|” – means or so ‘ro|bin’ matches ‘rbin’ or ‘roin’
  - [^] – when a class starts with ^ it means match anything BUT what is in the class so ‘[^gsf]+’ matches ‘robin’ ‘little’ ‘popocatepetl’ but not ‘great’ or ‘foo’ or ‘green socks are fun’

Regular Expressions

- Write a regular expression to match
  1. “integer”
     - integer
  2. “/*” or “*/”
     - ‘*?\/*?’
     - ‘(*\/)|(\/*)’
  3. A variable name
     - [a-zA-Z][a-zA-Z0-9_]*
  4. A string constant
     - \”.*\”

Regular Expressions

- Check out
  http://flex.sourceforge.net/manual/Patterns.html#Patterns

  for a reference on regular expressions in Flex
- Symbols to know: [], (), *, +, ?, ., |

Regex Cheatsheet

- http://guavus.files.wordpress.com/2009/05/regular-expressions-cheat-sheet-v2.png

A Silly Example

- silly[0-9]+ {adjust(); return SILLY;}
- moresilly[0-9]+ {adjust(); return SILLIER;}
- [silly0-9]+ {adjust(); return NOT_SILLY;}

- Which would match the following?
  - silly1
  - moresilly
  - silly

Start Conditions

- Start conditions are used to group rules that only apply at certain conditions
  - `\”              { BEGIN(STRING); }`
  - `<STRING>[^"]* { /* eat up the string body */}`
  - `<STRING>\"     { BEING(INITIAL); }`
- Rules without start conditions fall under INITIAL condition
- Two types of start conditions
  - EXCLUSIVE start conditions – only rules with that start condition are active
  - INCLUSIVE start conditions – rules with no start condition are ALSO active

Start Conditions

- Must be defined before rules section
- Defining start conditions
  - %s name – inclusive start condition
  - %x name – exclusive start condition
  - eg. %x GOOBER defines an exclusive start condition named goober
- Enter into a start condition with `BEGIN(<name>)`

Start Conditions

- Example (the actions are pseudocode):

```
ID [a-zA-Z_][a-zA-Z0-9_]*
%x COMMENT
%s NEWTYPE
%%
typedef          {BEGIN(NEWTYPE); return TYPEDEF;}
{ID}             {return IDENTIFIER;}
<NEWTYPE> {ID};     {BEGIN(INITIAL); return TYPENAME;}
"/*"             {IncreaseCommentCounter; BEGIN(COMMENT);}
<COMMENT>"/*"       {IncreaseCommentCounter;}
<COMMENT>"*/"       {DecreaseCommentCounter;
                 if (CommentCounter == 0){BEGIN(INITIAL);}}
<COMMENT><<EOF>>  {printf("Error: unclosed comment\n");}
"*/"  {printf("Error: unmatched \"*\/\" in line %d\n",
   yylineno);
```

Flex References

- Some sites to check if you are having problems
  - Flex in a nutshell-
  - http://lcs.syr.edu/faculty/mccracken/cis631/materials/04-Lex-In-A-Nutshell.pdf
  - Flex manual - http://flex.sourceforge.net/manual/index.html

Today’s lab – XML scanning

- Create a scanner that syntactically checks a given XML file and prints number of total elements found in it
- Must catch
  - Element mismatch: <a><b></a></b>
  - Non-closed elements <a><b></b><<EOF>>
  - XML syntax errors: <a id=“123” <b>>, <a><b<<EOF>>
- Must support
  - Elements with children: <a><b></b></a>
  - Elements with no children: <a/>
  - Element attributes: <a id=“123”/>

Today's lab

- When DEBUG is on, it must print the tokens as it scans
  - DEBUG is #define'd in the preamble

```
<a>
  <b>bbbb</b>
  <c>cccc</c>
  <d>
    <e f="g"/>
  </d>
</a>
```

$\longrightarrow$

```
<  (a)  >
  <  (b)  > TEXT(bbbb) </  (b)  >
  <  (c)  > TEXT(cccc) </  (c)  >
  <  (d)  >
    <  (e)   (f)=(g)  />
  </  (d)  >
</  (a)  >
# elements in the file
test_small.xml is 5
```

XML Rules

- Allowable element/attribute names:
  - `[a-zA-Z][a-zA-Z0-9:._]*`
  - `<abcd_1234 id1="a123"/>`: GOOD
  - `<@#$ id$="a123"/>`: BAD
- Attributes must have values
  - `<a id="123"/>`: GOOD
  - `<a id/>`: BAD

Today’s lab

- Checking for mismatch
  - Push name of the every new element onto a stack and pop them off as they are closed

```
<a>  <- push(‘a’);
  <b>  <- push(‘b’);
  </b> <- name=pop();  ‘b’ ?= name;
</a> <- name=pop();  ‘a’ ?= name;
```

Today's lab

- Modify xml.lex and add your rules
- To compile:
  - make
- To test:
  - ./scan test_good.xml
- Turn in any file you modify or the entire folder as a zip

Common Flex Errors

- You need to escape < > / and any other special character
  - `\<    \>    \/`
- "Rule cannot be matched" – you have another rule above this rule that is catching the token first
- Infinite Loops – make sure your <<EOF>> rules return(0);