More Examples of Flex

Lembit Jürimägi

Flex source file structure

- 4 sections:

  top

  definitions / options

  %%

  patterns / rules

  %%

  code

- Some sections may be empty

Flex file

```
%top{
#include "stdio.h"
}
%option case-insensitive
%option noyywrap
NUM    [0-9]
%x     STR
%%
{NUM}+       { printf("This looks like an integer: %d\n", atoi(yytext)); }
"\""         { BEGIN(STR); }
<STR>[^\"]+  { printf("quoted text: %s", yytext); }
<STR>"\""    { BEGIN(INITIAL); }
%%
int main(void){
       return yylex();
}
```

Flex: useful stuff

Options

- %option noyywrap – stops after scanning current file
- %option prefix="smth" – renames functions and variables from yy* to smth*
- %option case-insensitive – scanner is case insensitive
- %option yylineno – counts linenumbers, must have a rule for \n to function properly
- %option warn – prints warnings
- %option stack – allows explicit manipulation of states via a stack

Variables / Functions

- FILE *yyin – input file, by default set to stdin
- char *yytext – the currently matched input
- int yylineno – the line number in current input file
- int yylex() – starts the scanning process, returns when return statements present in rules or end of input
- BEGIN(); – macro for explicit machine state
- INITIAL – starting state
- unput() – puts character back into stream to be scanned next

Error reporting

- Scanner just finds matches to the defined patterns
- Errors are likely at higher abstraction, for example:
  - input doesn't match the context-free grammar rules of the programming language
  - integer constant is too large
- Still, when errors occur, the should be reported and user should find them easily
- yylineno is keeping track what line of the input file is currently being scanned
- A separate rule that matches newlines is necessary for it to work tho, even if it is:

```
\n      ; //do nothing
```

More than one scanner

- It may be necessary to scan different file types
  - For example, C files and ASM files
- The prefix option enables you to have several scanners in your project
- All the yy.* functions get renamed with the specified prefix replacing yy

For example:

```
%option prefix="asm_"
  ... atoi(asm_text) ...
  asm_in = fopen("code.s");
  asm_lex();
print("Error at line %d", asm_lineno);
```

Filename will be: lex.asm_.c

Start conditions / states

- In some cases it is easier to specify a separate case for handling some patterns
- For example, text string with escaped characters, we want to get rid of these and replace them with real character codes
- This may mean that instead of having a pattern match the entire string, we have to build up the string 1 character at a time.

```
%x STR
\"              pos = 0; BEGIN(STR);
<STR>[^\"]      buf[pos++] = yytext[0];
<STR>\\n        buf[pos++] = '\n';
<STR>\"         buf[pos++] = 0; BEGIN(INITIAL);
```

Inclusive / exclusive states

- %x specifies exclusive state
- This means that only rules that specify that state are used while in that state
- %s specifies inclusive state
- This means that rules that don't specify a state are applied while in this state

```
%x STR
%s ST2
\"      BEGIN(STR)
\'      BEGIN(ST2)
\n      ; //this applies for ST2 but not for STR
```

Stack

- %option stack makes it possible to use a stack of states
- Using stack lets us scan a language that is more complex than a regular language
- For example, we can check whether we have equal number of opening and closing parantheses

```
%s PAR
\(              yy_push_state(PAR);
<PAR>\)         yy_pop_state();
<PAR>\n         printf("too many opening\n");
<INITIAL>\)     printf("too many closing\n");
```

Context-free Languages

Lembit Jürimägi

Context-free Grammar

Context-free Grammars have following restrictions placed on production rules:

- A single nonterminal element allowed on the left side
- On the right side a rule is allowed to have
  - nothing
  - any number of terminal and/or nonterminal elements

Context-free Language

- Language generated by context-free grammar
- Language parser can be implemented using pushdown (stack) machine
- Slower parsing than regular languages
- Rules allow construction of complex enough structures for any programming language
- Rules are still too simplistic for describing natural languages
- Can be used as a parser for programming languages

Example: Mathematical Equations

G = {S, N, T, R}

- N : {S, A}
- T : { ( ) + - * / num var }
- R :
  - S -> A
  - A -> num
  - A -> var
  - A -> (A)
  - A -> A + A
  - A -> A - A
  - A -> A * A
  - A -> A / A

Postfix, infix, prefix calculator

- "fix" refers to the position of the operator
  - Postfix: 2 2 +
  - Infix: 2 + 2
  - Prefix: + 2 2
- Infix requires operator precedence and parentheses to overrule precedence
- Postfix and prefix don’t and can be parsed without look-ahead
- As long as we don't need look-ahead, we can use Flex