Lemon and RE2c

August 3, 2015

Lemon is different than Bison

- While there are many excellent guides to Bison, both commercial and free, Lemon has far fewer: the reason is simple, Lemon has far fewer quirks and oddities to document.

First things first

- With Lemon, the parser calls the lexer, instead of the very odd "lexer calls parser" structure that Bison and Flex use
- With Lemon, you don't write rules using a large number of alternatives in individual rules; rather, you split separate rules into separate cases.

No embedded semantic actions

- All semantic actions come *after* the rule, not in the midst.

Attributes

- If a component needs an attribute, you just use this simple syntax to name the attribute:

```
vardecl ::= VAR IDENTIFIER(ID_NAME) EQUALS expression.
{
  avl->add(avl,ID_NAME,"");
}
```

Attribute type declarations

- You declare its type in the %type section:

```
%type    IDENTIFIER        {char *}
```

So, what if I *need* an embedded semantic action?

- If you need the equivalent of an embedded semantic action, all you need to do is split the original rule into two parts: the original, and a "singleton" rule so that *its* reduction triggers your semantic action.

So, what if I *need* an embedded semantic action?

- For instance, let's say that you have an "if" construct something like this:

```
if_stmt ::= IF expr stmt_blk ELSE stmt_blk.
```

So, what if I *need* an embedded semantic action?

- and you would like to create some jump labels for the two branches. It would be nice to have these after the “if” is recognized, so split this one:

So, what if I *need* an embedded semantic action?

```
if_stmt ::= if expr stmt_blk ELSE stmt_blk.
if(JUMP_LABELS) ::= IF.
{
 // create two jump labels for the two cases
 JUMP_LABELS = create_two_jump_labels();
}
```

Handy debugging feature of Lemon

- ParseTrace()

You can have the parser emit very useful information about the state of the parse by simply calling the function ParseTrace()

```
ParseTrace(stderr, "PARSER SAYS: ");
```

Ending it all

- Don't forget to send the Parser a final "null" token:

```
Parser(Parser,0,0);
```

RE2C

- RE2C's model is quite different Lex/Flex; instead of writing a fairly complete lexer, RE2C expects you to (largely) write the structure of your lexer.

RE2C

- Inside of that structure, you insert comments with semantic meaning; these comments are *rewritten* by RE2C to provide fast DFA code to recognize those regular expressions.

RE2C, your code

```
/*!re2c

  re2c:define:YYCTYPE = "char";
  re2c:define:YYLIMIT = last_char;
  re2c:define:YYCURSOR = current_char;

  [ ... ]

  "," { return(","); }

*/
```

RE2C, after being rewritten

```
#line 120 "lexer.c"
    {
        char yych;

        yych = *current_char;
        switch (yych) {
        case ',':   goto yy2;
```