Creating your own Domain Specific Language (part 2)

Rolfe Bozier

28-May-2014

Agenda

- In this episode, we will cover:
  - Quick recap from last time
  - Creating a tokenizer using lex
  - Creating a parser using yacc
  - Fixing up problems in your grammar
  - Creating a syntax tree
  - Error handling

Quick recap

- A Domain Specific Language (DSL) can be a powerful tool for improving people's efficiency
  - DSLs are usually customised to a specific area of application
  - They aim to reduce the "impedence mismatch" between what you want to do and the task of carrying it out
- We are creating a language to test compositing operations in a Xebra renderer (ARR)
- We've got a bit of an idea what the language will look like
- We want to create a tool to parse the language, and hook it up to the UDI API

Our compositing language

```
let col1 = 100
let col2 = 200
let col3 = 300

let colours = [
    RGBA8 ff 00 00 ff,
    RGBA8 ff ff 00 ff,
    RGBA8 00 ff 00 ff,
    RGBA8 00 ff ff ff,
    RGBA8 00 00 ff ff,
    RGBA8 ff 00 ff ff,
]

let BLACK   = RGBA8 00 00 00 ff
let WHITE   = RGBA8 ff ff ff ff
```

```
newpage A4 RGBA8

loop
(
    c in colours;
    y in 100 step 20
)
    paint rect 10 10 at col1 y
        flat BLACK rop2 multiply
        flat c

    paint rect 10 10 at col2 y
        flat WHITE rop2 multiply
        flat c

    paint rect 10 10 at col3 y
        flat RED rop2 multiply
        flat c
endloop
```

Tools

- The tools we are going to use are available on Linux and probably Cygwin/Windows as well
  - **lex** (a.k.a. flex) manages tokenizing your input file
  - **yacc** (a.k.a. bison) matches the token stream with the rules in your grammar
- There are similar tools in other languages
  - PLY for Python
  - Perl::Yapp for Perl

lex

- **lex** is a tokenizer generator
  - It accepts a description of the lexical elements in a language (keywords, symbols, operators, variables, constants etc)
  - It generates C code that turns input text into a stream of tokens
  - Tokens are the *terminals* in your grammar
    - Some tokens have a value, e.g. INTEGER + <value>
    - Others are just IDs, e.g. NEWPAGE, EQUALS
  - The lex configuration file contains a set of regexps/strings, and a corresponding fragment of C code that is executed when a match is made

Example of a lex specification

```
#.*                    {}  /* ignore comments */
[ \t\r\v]              {}  /* ignore whitespace */
\n                     { lineno++; }

[...]

let                    { return LET; }
newpage                { return NEWPAGE; }
"="                    { return EQUALS; }

[...]

[0-9][0-9]*            { yylval.integer = atoi(yytext); return INTEGER; }
[_a-zA-Z][_a-zA-Z0-9]*    { yylval.id = strdup(yytext); return ID; }
```

Hints when using lex

- If more than one pattern matches…
  - the first is used, but… consider the following:

```
"a2"              { return A2PAPER; }
"a3"              { return A3PAPER; }
"a4"              { return A4PAPER; }

[...]

[0-9a-f][0-9a-f] { yylval.hex = strtohex(yytext); return HEXBYTE; }
```

- How would the following text be tokenized:

```
colour = ff cd a4 00;
```

```
VARIABLE ("colour")
HEXBYTE (0xff)
HEXBYTE (0xcd)
A4PAPER
HEXBYTE (0x00)
SEMICOLON
```

Hints when using lex

- You should probably just ignore white space and comments in your lexer
  - But keep track of the current line number, you'll need it for error reporting!
- Watch out for patterns that match very large chunks of text
  - You will overflow lex's buffer
  - But there is a work-around for this

yacc

- **yacc** is a parser generator
  - It accepts a BNF description of your grammar
  - It generates C code that matches a token stream against the rules in your grammar
  - Each production (rule) in the grammar has an associated fragment of C code (action)
  - The terminals and non-terminals in the grammar can have an associated value; the C fragments allow you to manipulate the value as a rule is matched

Example of a yacc specification

```
layer_list              : layer
                            { $$ = $1; }
                        | layer_list layer
                            { APPEND($$, layer_t, $1, $2); }
                        ;

layer                   : FLAT val ROP2 rop2type
                            { $$ = new_flat_layer($2, $4); }
                        | FLAT val
                            { $$ = new_flat_layer($2, ROP2_OVER); }
                        ;
```

- $$ refers to the attribute of the LHS
- $1, $2 etc. refer to the values of elements on the RHS
- Values for grammar elements are defined in a C union
  - Define the union member to associate with each non-terminal and terminal (optional)

How yacc works

- Yacc parsers use shift/reduce parsing
  - Tokens are read in and appended to a stack
  - When the elements on the stack match plus the current token a match grammar rule, they are popped off and replaced with the corresponding non-terminal (this is a *reduce*)
  - Otherwise the token is added to the stack (this is a *shift*)
  - Note that the parser only looks 1 token ahead to decide if it needs to reduce or shift
- You can create a grammar where yacc cannot decide between two or more possible actions
  - If it is ambiguous for yacc, it is probably ambiguous for users as well

Common yacc problems

- A *shift/reduce conflict* is where yacc could perform either action for the token stream:
  - Which is the correct parse for the following?

```
if (condition1)
then
    if (condition2)
    then
        action1
    else        ↑
        action2
```

```
if (condition1)
then
    if (condition2)
    then
        action1
else            ↑
    action2
```

- A *reduce/reduce conflict* is where yacc has multiple reductions available
  - Presumably the ambiguity could be resolved if yacc could look further ahead (but it can't…)

Beyond parsing the input

- Once you can parse a language, you need to able to do something with the parser
- The usual process is to create a *syntax tree* – a data structure that represents the whole of your program
  - I recommend dumping your syntax tree in readable form – it makes debugging easier!
- Once you have your syntax tree, you need to traverse it, performing all the appropriate actions for each node
- Real compilers do all their optimisation by transforming their syntax trees (but you won't be doing that!)

A quick note on error handling

- Error handling is a bit of a black art
- Yacc will flag any syntax errors and call a user-defined function – yyerror()
- You can raised your own errors when there are semantic violations (e.g. type mismatch errors)
- For simplicity you should just print the error and exit
  - You can add the special “error” non-terminal in selected places in your grammar – yacc will shift tokens until it can match an error-reduction
  - But… it’s hard.
  - and you’ve got an invalid syntax tree
  - and you may recover incorrectly because the wrong token was added, not removed.
- There is a good reason why compilers can give cascading and incorrect errors when faced with syntax errors ☺

Wrapping this part up

- We’ve used lex and yacc to parse our programs and generate a syntax tree
- Next time we’ll turn the tree into a set of API calls and hopefully have a working demo!
- There is more detailed information on the SEMG wiki page, along with the full source code for the lex and yacc files
  - I’ve also added in lex and yacc source for a C parser I wrote a number of years ago, to see what a full-blown language parser looks like