Creating your own Domain Specific Language (part 3)

Rolfe Bozier

10-Jun-2014

Agenda

- In the final episode, we will cover:
  - Quick recap from last time
  - Creating a syntax tree
  - Processing the syntax tree
  - The demo implementation
  - Final thoughts

Quick recap

- We have designed our language
- We have created a parser for the language, using lex and yacc
- We need to create and process the Abstract Syntax Tree (AST)
- We need to perform the appropriate actions for the program (render output)

Creating the AST

- The AST is just an efficient representation (C data structure) of our program
  - A program consists of a list of statements
  - A statement is one of:
    - A declaration statement (a name and a value)
    - A newpage statement (the paper size and colour space)
    - A loop statement (one or more iterators and a statement list)
    - A paint statement (a path and one or more layers)
  - An iterator is a variable name and a value list
  - A layer is a colour and a raster operation (ROP)
  - A value is an integer, a colour or a variable name

Creating the AST

- The AST data structures are built up by the *actions* in the yacc grammar.
- The association between the actions and the AST construction is managed by the %union, %token and %type declarations
- All data types that are associated with terminals or non-terminals are listed in the global %union definition:

```
%union
{
    char          *id;
    int           integer;
    char          *paper;

    stmt_t      *statement;
    iterator_t  *iterator;
    shape_t     *shape;
    layer_t     *layer;
    csspec_t    csspec;
    colour_t    colour;
    val_t       *val;
    rop_t       rop;
}
```

Creating the AST

- Terminals that do not have a value are declared using a %token definition:

```
%token ROP2_MULTIPLY RGB8 RGBA8 L8 LA8 CMYK8 CMYKA8 K8 KA8
%token EQUALS OROUND CROUND SEMICOLON COMMA OSQUARE CSQUARE
```

- Non-terminals will generally have some associated value , using a %type declaration:

```
%type <statement>   statement statement_list
%type <iterator>    iterator iterator_list
%type <shape>       shape
%type <layer>       layer layer_list
%type <val>         anyval val_list val_array val
%type <colour>      colourval
%type <integer>     intval
%type <csspec>      csspec
%type <rop>         rop2type
```

Processing the AST

- Originally, the goal was to embed our parser in a Xebra build and have it act as a new PDL interpreter:
  - Register as PDL interpreter
  - Sniff incoming files and “claim” if they are ours
  - Parse and call UDI functions:
    - UDI_document_start()
    - UDI_drawing_start()
    - UDI_object_paint()
    - UDI_drawing_finish()
    - UDI_document_finish()

Processing the AST

- Time was short, so a quick switch to plan B:
  - Generate a PDF file and feed this into a standard Xebra solution
  - Hide the details with some scripting ☺
- There are some disadvantages to this:
  - It's definitely not as easy to use
  - It limits functionality to that supported by PDF
  - It requires two passes through the AST
- So this would only be a stop-gap measure

Traversing the AST

- So… back to traversing the AST
  - For PDF output we need to go through the paint calls twice (once to create graphics state and once to do the drawing) – so we create a Drawing List
  - The traverse function is recursive
  - We need a symbol table for variables and iterators
    - Variables are global and immutable
    - Iterator scope is the loop body; update on each pass
  - After traversal we use the drawing list to create a PDF file

The demo code

File	Purpose
common.h	Common data structures and function declarations
tokens.l	The lex source code (tokenizer)
grammar.y	The yacc source code (parser)
execute.c	AST traversal and PDF generation
symtable.c	The symbol table for variables and iterators
composer.c	The main function

- Let's have a bit of a look at an actual example
- The source code will be available on the SEMG presentation pages
  - It should compile straightforwardly on a Linux box that has yacc and lex on it (some so, some don't)

What's missing or broken?

- There is no context for errors when traversing the AST:
  - `"error: expected colour value, found int"`
- The AST implementation for arrays is not quite right - you can't have an array of variables.
- There is no way to terminate a loop with just an integer iterator.
- There are no arithmetic expressions. They will probably be needed.
- The language syntax is a bit too minimalist, especially as goal was to be readable. For example, the following is a bit obtuse:
  - `"paint rect 10 50 at 80 y    flat bg"`
- There are no array dereferences - maybe this will be needed.
- A lot of these things will crop up when you start writing programs in your new language. So, don't release version 1 on your users...

In conclusion

- We have designed and implemented a prototype drawing language for testing compositing operations. It is [deliberately] restricted, but is easily extended.
- You could probably implement something similar is around a week (I certainly didn't spend that much time on this exercise...)
- The work was broken into three separate phases: language design, parser implementation and AST processing. In general each part is separate; I didn't touch the grammar or parser code at all while I implemented the AST traversal code.
- You should not expect your first version to be the final version. You'll never get it right first go.
- You will need to spend some time writing programs in your language, or find some very accepting users!
  - Try to stay faithful to your original language goals
  - Avoid adding features that C, Python, Perl have just because you can
  - Think about ease of use - consider adding some "syntactic sugar" to make it easier to read the programs (it will probably make your parser simpler as well!)