Creating your own Domain Specific Language

Rolfe Bozier

13-May-2014

Agenda

- We will cover:
  - an introduction to Domain Specific Languages (what, why, what [again], how)
  - defining the grammar of your language
  - using a parser generator to implement the grammar
  - adding the semantics to the parser
  - hooking it up to the real world
- We'll work through the creation of a simple language to support compositing testing
- It'll be a bit exciting because I haven't done the next part yet ☺

What is a Domain Specific Language?

- A language created to address a specific need, with functionality tailored to that need
- The syntax and semantics usually closely tied to a specific problem area
- Use often limited to people/organisations/teams working in a local domain
- Examples:
  - PDF, PostScript (printing)
  - HTML, Javascript (web pages)
  - TeX / LaTeX (document creation)
- Distinct from:
  - general purpose languages (C, Python, Java etc.)
  - file formats (TIFF, XML, etc.)

Why create a new DSL?

- The language can be closely aligned with the problem space
  - The gap between thinking about a problem and implementing a solution is small
  - You can try to eliminate the effort in converting the problem in your head into a general programming language
- You want to efficiently implement many solutions to a set of problems
- Allow domain experts (not programmers) to implement solutions
- The current tool sets do not provide efficient solutions
- Make things easier to reuse / revisit / rework
- You want to hide unimportant details from users
- You want to improve people's efficiency

What can you do with a new DSL?

- Convert programs to executable code, e.g.
  - direct to assembler / object code
  - to JVM bytecode
  - to object code via an intermediate language (e.g. C)
- Process programs into a new input for another process, e.g.
  - act as a pre-processor into an existing language
  - perform API calls
- Step through and "execute" the program directly
  - i.e. process it using an interpreter
- You can hide all the details of the conversion / processing steps with scripting

How would I create a new DSL?

- You have a problem space, and some tasks you want to perform
  - current solutions are unwieldy, time-consuming, unpopular, unreliable etc.
- Steps
  1. Come up with the workflow you want to support
  2. Design your language
  3. Create a parser for the language
  4. Validate / process the parser results
  5. Generate your output / perform your actions
- Most importantly, you want to leverage existing tools and techniques to achieve the above
  - There is no point in spending 6 months doing the above if it isn't going to save more time later on
  - Not to mention getting it past your manager!

Our problem space

- We want a new tool for creating compositing test cases
- Current options:
  1. Hand-craft a PDF file
     - Most people don't know much PDF
     - PDF is static content, so you cannot construct "actions"
  2. Create a UDI log using the udigen python library
     - More people know python
     - But udigen is a mapping of the UDI API, which can be a bit painful to use (it is stateless)
     - The "compositing" actions are obscured by the python code
- We want a new language that can clearly describe compositing operations
  - it should be more powerful than a static drawing language
  - the control mechanisms should not obscure the main actions
  - programs should be able to be rendered in our software

What is the desired workflow?

- The new language describes page content in terms of "patches" which contain flexible compositing actions
- The language parser will reduce the page description to a set of direct UDI API calls
- The UDI API implementation will be provided by ARR
- Processing programs written in our language will create the following results:
  - a TIFF image containing the results of the compositing actions
  - a text file that can be used to extract pixel colour values from the output image, for verification

What should the language look like?

- This is the more creative (and enjoyable) part of the process
- Your language must be closely aligned with people working in the problem domain
- There's a bit of theory behind languages and grammars, but provided you keep things reasonably simple the tools should just work for you
- Some things to avoid in your first language...
  - many data *types* (or maybe any types at all)
  - variables (or maybe make them generally immutable)
  - don’t have a multiple scopes
  - in general, don't add features unless you really need them
- Spend a bit of time thinking about things that could be ambiguous or painful to use
  - Once you've deployed a DSL, making changes becomes **much** harder!

Defining your language grammar - BNF

- You should document the language using [E]BNF ([Extended] Backus-Naur Form)
- BNF descriptions comprise:
  - *terminals* - literal values (strings, numbers, symbols, keywords)
  - *non-terminals* - a defined sequence of terminals and non-terminals
  - *productions* - rules that define non-terminals

Defining your language grammar - BNF

- Here is a simple example:

```
<primary-expression>  : INTEGER
                      | "(" <expression> ")"

<mul-expression>      : <primary-expression>
                      | <mul-expression> "*" <primary-expression>
                      | <mul-expression> "/" <primary-expression>

<add-expression>      : <mul-expression>
                      | <add-expression> "+" <mul-expression>
                      | <add-expression> "-" <mul-expression>

<expression>          : <add-expression>
```

- Note that:
  - operator precedence is built in
  - operator associativity is built in

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
```