C Programming Tools: Part 4

Building and Using your own Toolkit

Duncan C. White
d.white@imperial.ac.uk

Evangelos Ververas
e.ververas16@imperial.ac.uk

Dept of Computing,
Imperial College London

15th June 2017

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large.
- Some of our tools there - in particular, Datadec - were code generators - programs that write programs. Or as the Pragmatic Programmers put it: Write Code that Writes Code (Tip 29).
- Such tools defined some Little Language or Domain Specific Language to make our lives easier, and then translated that into (say) valid C code.
- Today, in the last C Programming Tools lecture, we'll find how to make writing code generators for little languages even easier.
- Specifically, by using Parser and Lexer Generator tools: Yacc and Lex.
- As always, there's a tarball of examples associated with this lecture. The handout and tarballs are available on CATE and at: `http://www.doc.ic.ac.uk/~dcw/c-tools-2017/lecture4/`

- Whenever you define a little language and want to write a code generator for it, the first step is writing parsers and lexical analysers. This problem has been solved! Lex and Yacc generate C code from declarative definitions of tokens and grammars.
- As a simple example, consider integer constant expressions such as `3*(10+16*(123/3) mod 7)`. The basic 'tokens' needed are:
  - Numeric constants (eg '123').
  - Various one-character operators (eg. '(', '+', '*', ')' etc).
  - A Haskell-inspired keyword 'mod' (i.e. modulus, '%' in C terms).
- With Lex, specify the tokens as regular expression/action pairs:

```
[0-9]+              return NUMBER;
\+                  return PLUS;
-                   return MINUS;
\*                  return MUL;
\/                  return DIV;
mod                 return MOD;
\(                  return OPEN;
\)                  return CLOSE;
[ \t\n]+            /* ignore whitespace */;
.                   return TOKERR;
```

- See lexer.l for the full Lex input file, containing the above plus some prelude. This file can be turned into C code via: lex -o lexer.c lexer.l.

- These tokens can be combined to form expressions using the following BNF-style grammar rules (in Yacc-format):

```
%token PLUS MINUS MUL DIV MOD OPEN CLOSE TOKERR
%token NUMBER

%start here
%%
here        : expr
            ;
expr        : expr PLUS term
            | expr MINUS term
            | term
            ;
term        : term MUL factor
            | term DIV factor
            | term MOD factor
            | factor
            ;
factor      : NUMBER
            | OPEN expr CLOSE
            ;
```

- parser.y contains these rules plus some Yacc-specific prelude, including a short main program that calls the parser. This can be turned into C code (parser.c and parser.h) via: yacc -vd -o parser.c parser.y
- You can now compile and link parser.c and lexer.c to form expr1, just type make. See the Makefile for details. expr1 is a recognizer: it will say whether or not the expression (on standard input) is valid.

- Directory 02.expr2 extends our recognizer so that it calculates the value of the expression and displays it. There are two sets of changes from the previous version:
- First, we modify one line in lexer.l to store the integer constant value into 'yylval.n':

```
[0-9]+                          yylval.n=atoi(yytext); return NUMBER;
```

- Second, in parser.y there are several changes: add to the prelude:

```
static int expr_result = 0;
```

  Then make main display the result after a successful parse:

```
printf( "result: %d\n", expr_result );
```

- Above the token definitions, add:

```
%union { int n; }
%token <n> NUMBER
%type <n> expr term factor
```

- Add actions to grammar rules taking the calculated value from each sub-part and computing the result, plus a top level action which sets expr_result. Here's a sample:

```
here        : expr              { expr_result = $1; }
            ;
expr        : expr PLUS term    { $$ = $1 + $3; }
            | expr MINUS term   { $$ = $1 - $3; }
            | term              { $$ = $1;      }
            ;
term        : term MUL factor   { $$ = $1 * $3; }
            | term DIV factor   { $$ = $1 / $3; }
            ...
```

- After make we have expr2, an expression calculator. Play with it.

- Directory 03.expr3 extends our expression language, allowing a factor to be an identifier - an IDENT token - representing a named constant. There are three sets of changes from the previous version:
- Add a new consthash module, which stores our named constants.
- Add a line in lexer.l to recognise and return our new token:

```
[a-z][a-z0-9]*          yylval.s=strdup(yytext);return IDENT;
```

- parser.y has several changes: add to the prelude: `#include "consthash.h"`
  Then main() needs to create the constant hash right at the start, destroy it at the end:

```
init_consthash( argc, argv );
if( yyparse()....
destroy_consthash();
```

- Change the union declaration to: `%union { int n; char *s; }`
- Declare that the IDENT token has an associated string value:

```
%token <s> IDENT
```

- Add the new factor rule:

```
            | IDENT             { $$ = lookup_const($1); }
```

- After make we have expr3, a calculator with named constants. Play with it.

- Directory 05.expr5 contains our final Yacc/Lex expression example, which replaces calculation with treebuilding (using Datadec). Prepare types.in file:

```
TYPE {
  arithop =  plus or minus or times or divide or mod;
  expr    =  num( int n )
          or id( string s )
          or binop( expr l, arithop op, expr r );
}
```

- Alter the Makefile to invoke datadec generating types.c and types.h. parser.y has several changes: add to the prelude: `#include "types.h"`
- Change expr_result from an int to an expr: `static expr expr_result = NULL;`
- main should print out the expression tree (on parse success):

```
print_expr( stdout, expr_result );
```

- Change the union declaration to: `%union { int n; char *s; expr e; }`
- Change the type of all expression rules to e, the union's expr:

```
%type <e> expr term factor
```

- Change all the actions, for example:

```
expr        : expr PLUS term  { $$ = expr_binop( $1, arithop_plus(), $3 ); }
            | expr MINUS term { $$ = expr_binop( $1, arithop_minus(), $3 ); }
       ...
factor      : NUMBER          { $$ = expr_num($1); }
            | IDENT           { $$ = expr_id($1); }
```

- After make we have expr5, an expression parser and treebuilder.

- Expressions are hardly impressive! But Yacc, Lex and Datadec easily scale to much larger languages.
- Define a tiny Haskell subset called THS, build a Lexer and Parser using Lex and Yacc, build an Abstract Syntax Tree using Datadec, with parse actions to build our AST.
- Ok, what Haskell subset? Specifically, we'll allow:
  - Zero-or-more function definitions, with optional type definitions,
  - Followed by a compulsory integer expression (often a call to one of those functions).
  - Each function takes and returns a single integer value,
  - Each function implemented either by a single expression, or
  - A sequence of guarded expressions involving simple boolean expressions, eg. `x==0`,
- For example:

```
f x = 1

abs x | x>0  = x
      | x==0 = 0
      | 0>x  = 0-x

f(20) + abs(10) * 30
```

- In a break with strict Haskell-syntax, we'll decide that brackets on function calls like `abs(10)` are compulsory.

- Note in passing that we reuse (and extend) our expression grammar rules – hence any valid expression is also a valid THS program, one with no function definitions.
- Ok, first we define our lexer rules, regexps and actions:

```
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
mod                     return MOD;
Int                     return INTTYPE;
True                    return TRUEV;
[a-z][a-z0-9]*          yylval.s=strdup(yytext);return IDENT;
::                      return COLONCOLON;
->                      return IMPLIES;
==                      return EQ;
=                       return IS;
>                       return GT;
!=                      return NE;
\+                      return PLUS;
-                       return MINUS;
\*                      return MUL;
\/                      return DIV;
\(                      return OPEN;
\)                      return CLOSE;
\|                      return GUARD;
[ \t\n]+                /* ignore whitespace */;
.                       return TOKERR;
```

- Note that we are being extremely minimal with our tokens, including (for example) True but not False. These can trivially be added.

- As usual, our grammar and (Datadec-generated) AST intertwine, let's start by looking at types.in - our Datadec input file:

```
arithop   =  plus or minus or times or divide or mod;
expr      =  num( int n )
             or id( string s )
             or call( string s, expr e )
             or binop( expr l, arithop op, expr r );
boolop    =  eq or ne or gt;
bexpr     =  truev
             or binop( expr l, boolop op, expr r );
guard     =  pair( bexpr cond, expr e );
guardlist =  nil
             or cons( guard hd, guardlist tl );
fdefn     =  onerule( string fname, string param, expr e )
             or manyrules( string fname, string param, guardlist l );
flist     =  nil
             or cons( fdefn hd, flist tl );
program   =  pair( flist l, expr e );
```

- In parser.y, here's our `%union` declaration, which lists all possible types of data associated with tokens and grammar rules:

```
%union
{
        int       n;    char      *s;
        expr      e;    bexpr      b;
        guard     g;    guardlist gl;
        fdefn     f;    flist     fl;
}
```


- Here are some of the declarations that associate tokens and grammar rules with specific members of the union:

```
%token <n> NUMBER
%token <s> IDENT
%type <e> factor term expr
%type <b> bexpr
%type <g> guard;
...
```

- Let's look at a few grammar rules to give a flavour:

```
program      :  defns expr      { prog_result = program_pair( $1, $2 ); }
             ;
defns        :  /* empty */     { $$ = flist_nil(); }
             | defns ftypedefn  { $$ = $1; /* ignore type defns */ }
             | defns fdefinition { $$ = flist_cons( $2, $1 ); }
             ;
ftypedefn    : IDENT COLONCOLON type IMPLIES type { free_string( $1 ); }
             ;
type         : INTTYPE
             ;
fdefinition  : IDENT IDENT IS expr { $$ = fdefn_onerule( $1, $2, $4 ); }
             | IDENT IDENT guardrules
               {
                        guardlist rightorder = reverse_guardlist($3);
                        $$ = fdefn_manyrules( $1, $2, rightorder );
                        free_guardlist_without_guard( $3 );
               }
             ;
guardrules   : guard           { $$ = guardlist_cons($1, guardlist_nil()); }
             | guardrules guard { $$ = guardlist_cons( $2, $1 ); }
             ;
...
```


- Note that recursive rules in Yacc, such as:

  ```
  guardrules   : guardrules guard
  ```

  must place the recursive invocation first, hence when we build the AST guardlist it's in the reverse order. To fix this, we defined our own reverse_guardlist() function in the prelude.
- I've attempted to free() everything I malloc(), checking with valgrind. The reversing exposes a shared pointers subtlety: we build a new guardlist with the same heads (guards) as the original list. We must only free each guard once!
- To fix this, we had to add free_guardlist_without_guard() to the prelude, and call it from the above Yacc action to free the original guardlist.
- free_guardlist_without_guard() is a copy of the automatically generated free_guardlist() function, with the free_guard(head) call commented out.
- Putting it altogether, adding named constants (via the hash module), using datadec and our macro tool from the previous lecture, we end up with a THS (Tiny Haskell subset) parser and treebuilder, of which we only write about 460 lines of code.
- Give it a try!

- 07.ths-codegen extends our treebuilder, adding semantic checking (eg. checking that we define every function we call) and then code generation - translating THS to C!
- How do we do semantic checks? A semantic checker involves walking the AST and building convenient data structures. We create a hash and a set: the hash maps from functionname to AST function definition (for every defined function); the set names all called functions. Then we check that every called function is defined, exactly once.
- How do we do code generation? A code generator is just another ASTwalker, one with suitable print statements!
- In fact, using datadec's print hints mechanism, 80% of the C code generation was done by making each AST type print itself in valid C form. The remaining 20% was custom C code, mainly printing boilerplate and then invoking datadec-generated print_TYPE() functions.

- We're now using so many tools to build our code, let's see what percentage of the source code we're writing manually.
- In 07.ths-codegen, we have only written about 900 lines of code ourselves.
- However, after datadec, macro, Yacc and Lex have run, there are approximately 5400 lines of C code (including headers) overall.
- 900/5400 is about 16%.
- To put that another way: *our tools wrote 84% of the code for us.*

Ok, let's sum up what we've been trying to say in these lectures:

- Follow 100,000 years of human history by tool-using and tool-making.
- Are we Homo sapiens - or Homo faber, man the toolmaker?
- Build yourself a powerful toolkit.
- Choose tools you like; become expert in each.

- When necessary and practical, build tools yourself to solve problems that irritate you. Don't be afraid!
- Tools may save you much more time than they cost you to make.
- Other possible tools I didn't mention: regular expression libraries; all the things you can do with function pointers; text processing tools; OO programming in C etc etc.
- Most importantly: enjoy your C programming! Build your toolkit - and let me know if you write any particularly cool tools!
- Scripting languages like Perl, Ruby or Python are fantastic timesavers. I used to run a Perl course until it got cancelled, notes available at: `http://www.doc.ic.ac.uk/~dcw/perl2014/`
- Finally, I've also written an occasional series of Practical Software Development articles, see: `http://www.doc.ic.ac.uk/~dcw/PSD/`
- That's all folks!