Building your own C Toolkit: Part 3

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

Dept of Computing,
Imperial College London

12th June 2014

- Last week, we continued the idea of building a C programming toolkit, covering:
  - Generating prototypes automatically: proto.
  - Fixing memory leaks: libmem.
  - Optimization and Profiling.
  - Generating ADT modules automatically.
  - Reusable ADT modules: hashes, sets, lists, trees etc.
- Today, we're going to finish off our C Tools lectures, and cover:
  - Parser and Lexer Generator tools: Yacc and Lex.
- As last week, there's a tarball of examples associated with this lecture. Both lectures' slides and tarballs are available on CATE and at: `http://www.doc.ic.ac.uk/~dcw/c-tools-2014/`

- Scaling the previous idea of little languages up, you often need to write parsers and lexical analysers. This problem has been solved! Like Datadec, Lex and Yacc generate C code from declarative definitions of tokens and language 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).
- Specify the input tokens as regular expressions:

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

- See lexer.l for the full Lex input file, containing the above rules and 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 oneexpr
%%
oneexpr      : 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 value of the integer constant 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 with more than one sub-part, 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:

```
oneexpr       : expr            { expr_result = $1; }
              ;
expr          : expr PLUS term   { $$ = $1 + $3; }
              | expr MINUS term  { $$ = $1 - $3; }
              | term
              ;
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 calculator, 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 > 1 );
if( yyparse()....
destroy_consthash();
```

- Change the union declaration to:

```
%union { int n; char *s; }
```

- Tell the parser that IDENT builds a string:

```
%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, add Makefile rules:

```
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 )
        ;
}
```

- 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.
- Let's define a tiny Haskell subset (called HS) build a Lexer and Parser using Lex and Yacc, build an Abstract Syntax Tree using datadec, then add parse actions to build our AST.
- Ok, what Haskell subset? Specifically, we'll allow:
  - Zero-or-more function definitions, with optional type definitions,
  - Taking and returning a single integer value,
  - Implemented either by a single expression, or
  - A sequence of guarded expressions involving simple boolean expressions, eg. `x==0`,
  - Followed by a compulsory integer expression (often a call to one of the functions defined earlier).
- 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 a function call 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 HS program, one with no function definitions.
- Ok, first we define our lexer rules, regexps and tokens:

```
[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;
\n                      /* ignore end of line */;
[ \t]+                  /* ignore whitespace */;
.                       return TOKERR;
```

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

- 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 */     { $$ = flist_nil(); }
             | defns ftypedefn    /* ignore type defns */
             | defns fdefinition { $$ = 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.
- New this year: having added experimental free_TYPE() support to datadec, I've attempted to free() everything I malloc() (using libmem to help out). 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.
- Finally, datadec has a feature I didn't mention last time, you can specify how to print each shape of each data type via print hints. Read datadec's man page, and look inside types.in to see how this works.
- Putting it altogether, adding named constants (via the hash module), and generating some boilerplate using our tiny tool from the first lecture, we end up with a HS (Haskell subset) parser and treebuilder. Give it a try!

- 07.hs-codegen extends our treebuilder, adding semantic checking (eg. checking that every function call is to a defined function) and then code generation - translating HS to C!
- How do we do semantic checks? A semantic checker involves walking the AST and building convenient data structures. We create two hashes: one maps from functionname to AST function definition (for every defined function); the other represents a set of 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.hs-codegen there are approx 5400 lines of C code (including headers), we wrote about 900 lines ourselves. That's about 16 %.
- Left for you: Remember Dafny from Sophia's first year logic lectures?
- 08.hs2dafny-codegen translates HS to Dafny for verification.
- The basic work we need to do is change the codegen treewalker and some of the print hints.
- In fact, I made a few extra changes to generate better Dafny code: added a few more boolean operators and an "otherwise" keyword, and sneakily overrode one of the datadec-generated print functions with one I wrote myself.
- I didn't have the time to add libmem checking to this version, feel free to have a go yourself.

- Follow 100,000 years of human history by tool-using and tool-making. Build yourself a powerful toolkit. Choose tools you like; become expert in each.
- When necessary, build tools yourself to solve problems that irritate you. Don't be afraid! Try to build tools that save you more time than they cost you to make.
- 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!
- Finally, scripting languages like Perl, Ruby or Python are fantastic timesavers. I run a Perl course each January, notes available at: `http://www.doc.ic.ac.uk/~dcw/perl2013/`