C Programming Tools: Part 5

Building Lexers and Parsers (cont)

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

Dept of Computing,
Imperial College London

- Last lecture, we started using Yacc and Lex along with Datadec, tmpl and proto to build a simple calculator for integer expressions.
- Now, in the last C Programming Tools lecture, we'll show how Yacc and Lex scale up to help build much larger Code Generators.
- Then we'll briefly discuss a recently discovered alternative parsing-based technique that I've been playing with.
- Then we'll finish off by summarising the Programming Tools philosophy.
- The handout and tarball are available on materials and at:
  `http://www.doc.ic.ac.uk/~dcw/c-tools-2021/lecture5/`

- It's time to change up a gear: our second example is for a Tiny Haskell Subset I imaginatively name THS.
- THS comprises:
  - Zero-or-more function definitions, with optional type definitions;
  - Followed by a compulsory integer expression (often a call to some of those functions);
  - Each function takes and returns a single integer value;
  - Each function is implemented either by a single expression, or
  - A sequence of guarded expressions involving simple boolean expressions, eg. `x==0`.
- For example:

```
double :: Int -> Int
double x = x*2
abs x | x>0  = x
      | x==0 = 0
      | 0>x  = 0-x
fact x | x==1 = 1
       | x>1  = x * fact(x-1)
double(20) + abs(0-2)*fact(arg1)
```

- In a break with strict Haskell-syntax, we'll decide that brackets on function calls like `abs(10)` are compulsory. Why? Because the lack of brackets confuses me:-)

- At the lexical level, we add the following new regex/action pairs to our Lex input file lexer.l - keeping all the integer expression rules unchanged:

```
Int                     return INTTYPE;
True                    return TRUEV;
::                      return COLONCOLON;
->                      return IMPLIES;
==                      return EQ;
=                       return IS;
>                       return GT;
!=                      return NE;
\|                      return GUARD;
```

- Note that we have included True but not False, '>' but not '<' etc. These can be trivially added later - in fact, this would make a good exercise for anyone who's interested.
- The Abstract Syntax of THS, in types.in, is more complex:

```
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 )
          or call( string s, expr e );
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 );
fbody     =  one( expr e ) or many( guardlist l );
fdefn     =  triple( string fname, string param, fbody b );
flist     =  nil or cons( fdefn hd, flist tl );
program   =  pair( flist l, expr e );
```

- Turning to the Yacc input file parser.y, Our ast variable (that stores the AST after a successful parse) was an expr, now it's

```
program ast = NULL;
```

- The `%union` declaration is much bigger this time:

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

- Our token lists are bigger than before:

```
%token COLONCOLON IMPLIES EQ GT NE TRUEV PLUS MINUS MUL DIV MOD OPEN
       CLOSE GUARD IS INTTYPE TOKERR
%token <n> NUMBER
%token <s> IDENT
```

- Our parse rule type association list is also bigger:

```
%type <e>  factor term expr
%type <b>  bexpr
%type <g>  guard
%type <gl> guards
%type <f>  fdefinition
%type <fl> fdefns
```


- Next, parser.y tells Yacc which parse rule to start parsing with:

```
%start program
```

  Recall that this forces the parser to parse the entire input using the program rule.
- The rest of the parser.y lists the grammatical parse rules that define THS, plus the corresponding tree-building actions to take when the rules match:

```
%%
program      : fdefns expr              { ast = program_pair( $1, $2 ); }
             ;
fdefns       : /* empty */              { $$ = flist_nil(); }
             | fdefns ftypedefn         { $$ = $1; /* ignore type defns */ }
             | fdefns fdefinition       { $$ = flist_cons( $2, $1 ); }
             ;
ftypedefn    : IDENT COLONCOLON INTTYPE IMPLIES INTTYPE { free_string( $1 ); }
             ;
fdefinition  : IDENT IDENT IS expr      { $$ = fdefn_triple( $1, $2, fbody_one($4) ); }
             | IDENT IDENT guards       { $$ = fdefn_triple( $1, $2, fbody_many($3) ); }
             ;
guards       : guard                    { $$ = guardlist_cons($1, guardlist_nil()); }
             | guards guard             { $$ = guardlist_push( $1, $2 ); }
             ;
guard        : GUARD bexpr IS expr      { $$ = guard_pair( $2, $4 ); }
             ;
bexpr        : expr EQ expr             { $$ = mkequals( $1, $3 ); }
             | expr NE expr             { $$ = mknotequals( $1, $3 ); }
             | expr GT expr             { $$ = mkgreaterthan( $1, $3 ); }
             | TRUEV                    { $$ = bexpr_truev(); }
             ;
```


- The grammar rules finish off with expr, term and factor, mostly unchanged, although there's an extra factor rule, allowing a function call:

```
factor     : IDENT OPEN expr CLOSE     { $$ = expr_call($1,$3); }
```

- Picking one of our parse rules/action pairs out for a detailed inspection, we see:

```
guard        : GUARD bexpr IS expr     { $$ = guard_pair( $2, $4 ); }
```

- When this rule matches, $2 is the abstract boolean expression and $4 is the abstract integer expression. The action builds `guard_pair( $2, $4 )`, and assigns it to $$.
- Having built a new abstract guard, and associated it with the successful match of the guard parse rule, parsing continues trying to parse a non-empty sequence of guards, in which we have either a single guard, or some guards followed by one more guard:

```
guards       : guard                   { $$ = guardlist_cons($1, guardlist_nil()); }
             | guards guard            { $$ = ?????( $1, $2 ); }
             ;
```

- When the guards guard rule matches, we want our action to build a guard list with the guards *in the order they were encountered* in the THS input file.

- But Yacc's love of left recursion causes us a problem, because:
  - $1 is set to the guardlist value associated with the guards recursive invocation (which has just matched all guards except the last one);
  - and $2 is set to the value associated with the last guard.
- So $1 is a guardlist, and $2 is a single guard (the last one). If we write the obvious action `$$ = guardlist_cons($2,$1)` we generate the guard list in reverse order, causing us a problem later on.
- Obviously, we can't swap the arguments and write `$$ = guardlist_cons($1,$2)` because the generated C code will not compile.
- In a previous version, I let Yacc build the guard list in reverse order, and then wrote a `guardlist_reverse()` function later.
- But now, the action I write is `$$ = guardlist_push($1,$2)`. This function was manually written (you'll find it in the prelude section of types.in) and modifies the existing guardlist, finding the last node and adding the new guard on the end.

- Once we have built our guard list, when an entire function with guard list body is parsed successfully, the guard list gets incorporated into an abstract function definition by the fdefinition parse rules and actions.
- Function definitions get incorporated into function lists, by the fdefns parse rules and actions. Note that function type definitions are simply discarded. But why do we have to `free_string($1)`?
- Note that the function lists are built in reverse order, because we build them using `flist_cons()`. This doesn't matter because the order of functions is irrelevant - unlike the order of guards in a function body which mattered.
- Finally, turning to the start rule, program:

```
program     :  fdefns expr           { ast = program_pair( $1, $2 ); }
            ;
```

  When this rule matches the entire input, the action is invoked with the final function list in $1, and the main expression in $2, both get incorporated into a program_pair(), which is assigned to program ast.

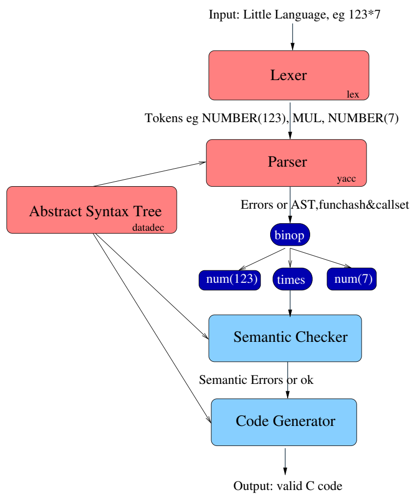
- The 01-ths.treebuilder directory also provides:
  - The main program (mainprog.c) roughly as it was in the expression example,
  - The consthash module exactly as it was from the expression example,
  - arithfuncs.in (replacing binfuncs.in), a tmpl-format input file generating the arithfuncs module using mkmodule, and
  - boolfuncs.in, a tmpl-format input file generating the boolfuncs module containing mkequals(), mknotequals() and mkgreaterthan().
  - A Makefile to generate and compile everything.
- However, the integer expression evaluator module has been removed - for now, we'll come back to this.
- Note also the types.in uses a useful Datadec feature we haven't discussed so far - the print hints mechanism whereby you annotate each shape of each inductive data type telling Datadec how to print it out. See if you can work out how it works.
- Compile and link by typing make. We end up with a THS parser and treebuilder ths1, in which we only write about 430 lines of code. Give it a try.

- 02.ths-semanticchecker adds semantic checking - in THS, this means checking that we define every function we call, and also that every use of an identifier inside an (integer or boolean) expression is either a predefined named constant in consthash, or the current function's parameter. In other languages, we'd have to perform other semantic checks - for example the number and types of actual parameters to each called function.
- How do we do semantic checks? A semantic checker either walks the AST, or builds and iterates over equivalent data structures.
- To reduce tree-walking, we enhanced parser.y as follows:
  - As we parse each function, we populate a hash called funchash, mapping the function name to it's abstract representation;
  - As we parse function calls, populate a set called callset - the set of all called functions.
  - For simplicity, we perform the identifier in a factor checks inside parser.y, via a new check_id() function. There's always a fine line between parse checks and semantic checks.
- After a successful parse, the semantic checker iterates through the callset checking that each called function is present in the

- 03.ths-interpreter extends our semantic checker, adding an interpreter to run our THS programs.
- How do we write the interpreter? Well, you've written interpreters in Haskell before, so the principles should be familiar:
- We write C functions to:
  - Evaluate an integer expression in the current environment.
  - Evaluate a boolean expression in the current environment.
  - Select which guard in a guardlist is true and then evaluate it's corresponding integer expression, all in the current environment.
  - Handle a function call (possibly recursive).
- The only tricky part is that in a function call, we evaluate the actual parameter expression in the current environment, giving the integer result X, then create a new environment in which the function's parameter variable is set to X, then evaluate the function body (expression or a guardlist) in the new environment.
- If we do this right, our interpreter will correctly handle recursion.
- Note that we also have to trap runtime errors such as division by zero and what happens if no guard evaluates to true.

- Our final version of THS, 04.ths-codegen, replaces our interpreter with a code generator - which translates THS to C!
- How do we do code generation? A code generator is just another AST and funchash walker, 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% (approx 60 lines) was custom C code, gluing everything together.
- One subtlety was that Haskell/THS allows any function to call any other function. This means that the generated C code needs a block of prototypes for all THS functions. This requires one more pass through the funchash, emitting a prototype for each THS function.
- Another subtlety was that we have to prevent a function falling off the bottom (when no guard evaluates to true).

They say a picture's worth a thousand words, so let's recap:



- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar, builds an AST and builds funchash and callset (not shown).
- Our Semantic checker uses the AST, funchash and callset to check that there are no consistency problems.
- Our Code generator walks the AST and funchash, emitting C code.

- 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 04.ths-codegen, the `make lines` target tells us that we have only written 777 lines of code ourselves - the Lex input file, the Yacc input file, the Datadec input file, various tmpl-format input files, and some C code (.c files and .h files).
- After **datadec, mkmodule, yacc** and **lex** have run, there are approximately 5200 lines of C code (including headers) overall.
- 777/5200 is about 14%.
- To put that another way: *our tools wrote 86% of the code for us*.
- That's pretty impressive - very few combinations of tools automate anywhere near that much of our code!
- So, Yacc and Lex and Datadec are a scalable way of building translators for little languages, vital tools for your toolbox.
- In the tarball, left for you to explore, there's an extended version of THS - that I call BHS (for "Bigger Haskell Subset") that allows functions to have multiple parameters - all still integer.

- Recently, I've been playing with a different parsing approach: Suppose instead of defining a complete little language, we want to add a single well-defined feature to a large language like C.
- For example: Datadec has no special support for writing client-side code that uses datadec-generated types. You may remember our tree type, and our nleaves() counter, from lecture 3. From time to time I've thought that some sort of shaped pattern match would be lovely - in C. I'd love to be able to write, in an enhanced C-like language:

```
int nleaves( tree t )
{
	whenshape t is leaf(name)
	{
		return 1;
	}
	whenshape t is node( l, r )
	{
		return nleaves(l) + nleaves(r);
	}
}
```

- Having defined the syntax of the new feature, we define it's semantics via a precise description of how to translate it back to standard C.

- The first whenshape example turns into the plain C code:

```
if( tree_kind( t ) == tree_is_leaf )
{
        string name; get_tree_leaf( t, &name );
        return 1;
}
```

- Similarly, the second whenshape example turns into:

```
if( tree_kind( t ) == tree_is_node )
{
        tree l; tree r; get_tree_node( t, &l, &r );
        return nleaves(l) + nleaves(r);
}
```

- But how do we implement this? In Yacc and Lex, we'd have to implement all of normal C as well as our new feature.
- We could get a complete open-source C compiler (like Gcc or Clang) and graft our new feature into it.
- But that sounds like hard work! Gcc is very complex.

- Another way would be to build (or find) a *C to C translator* which can be extended. Perhaps someone has already built one that we could extend?
- If not, you could build one by finding a complete Yacc grammar spec, Lex lexer spec and AST module for C and extend them - adding our new tokens to the lexer spec, new rules to the grammar spec to recognise our new forms of syntax, and new actions to build AST fragments representing the plain C equivalents for each new construct.
- This also sounds like a lot of work!
- Isn't there a ...smaller way to do this? That might be doable in an evening? Yes there is!
- Graft our new feature into C by writing a simple line-by-line pre-processor that copies most lines through unchanged (assuming, or hoping, that they contain valid C), but locates specially marked extension directives, turning each into a corresponding chunk of plain C.
- Thus, C with directives comes in, standard C goes out.

- In 05.c+pattern-matching you'll find my experimental Perl script cpm, which translates C with pattern matching to plain C, working in concert with datadec.
- In the tree-eg subdirectory, you'll find nleaves.cpm that implements a close approximation to what we wanted to write:

```
int nleaves( tree t )
{
        %when tree t is leaf(name)
        {
                return 1;
        }
        %when tree t is node( l, r )
        {
                return nleaves(l) + nleaves(r);
        }
}
```

- There are several other pattern matching directives as well.
- See interprete.cpm (found in the `interprete-eg` subdir) for a bigger example - the THS interpreter rewritten using the lovely new syntax.
- BTW, cpm reads information about types, shapes, and their parameters from datadec in a particularly sneaky fashion, which I'm very proud of.

Ok, that's quite enough parsing. Let's sum up what I've been trying to say in these lectures - the Programming Tools philosophy:

- 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. Be strong! Tools often save you much more time than they cost you to make.
- Text manipulation languages are fantastic timesavers. Perl is especially good - known as The Swiss Army Chainsaw by SysAdmins. I used to run a Perl course, see `http://www.doc.ic.ac.uk/~dcw/perl2014/`
- I also write an occasional series of Practical (Pragmatic?) Software Development articles: `http://www.doc.ic.ac.uk/~dcw/PSD/`
- Read The Pragmatic Programmer. Then read it again!
- Most importantly: enjoy your C programming! Build your toolkit - and let me know if you build any particularly cool tools!