C Programming Tools: Part 4

Building and Using your own Toolkit

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

Dept of Computing,
Imperial College London

14th June 2018

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large.

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.
- Today, in the last C Programming Tools lecture, we'll find how to make writing Code Generators even easier.

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.
- Today, in the last C Programming Tools lecture, we'll find how to make writing Code Generators even easier.
- The first part of writing any Code Generator is to build a lexical analyser (aka a lexer) and a parser for your little language. It's instructive to write a couple of lexers and parsers by hand to get the hang of it, but..

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.
- Today, in the last C Programming Tools lecture, we'll find how to make writing Code Generators even easier.
- The first part of writing any Code Generator is to build a lexical analyser (aka a lexer) and a parser for your little language. It's instructive to write a couple of lexers and parsers by hand to get the hang of it, but..
- This problem has been solved!

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.
- Today, in the last C Programming Tools lecture, we'll find how to make writing Code Generators even easier.
- The first part of writing any Code Generator is to build a lexical analyser (aka a lexer) and a parser for your little language. It's instructive to write a couple of lexers and parsers by hand to get the hang of it, but..
- This problem has been solved! Lex generates C code (a lexer) from declarative definitions of lexical tokens.

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.
- Today, in the last C Programming Tools lecture, we'll find how to make writing Code Generators even easier.
- The first part of writing any Code Generator is to build a lexical analyser (aka a lexer) and a parser for your little language. It's instructive to write a couple of lexers and parsers by hand to get the hang of it, but..
- This problem has been solved! Lex generates C code (a lexer) from declarative definitions of lexical tokens. Yacc generates C code (a parser) from declarative definitions of the grammar, plus actions to take when grammatical constructs are parsed successfully.

- Last week, we started building our own tools when necessary, at a range of scales from tiny to large. Those tools were Code Generators - Code that Writes Code.
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.
- Today, in the last C Programming Tools lecture, we'll find how to make writing Code Generators even easier.
- The first part of writing any Code Generator is to build a lexical analyser (aka a lexer) and a parser for your little language. It's instructive to write a couple of lexers and parsers by hand to get the hang of it, but..
- This problem has been solved! Lex generates C code (a lexer) from declarative definitions of lexical tokens. Yacc generates C code (a parser) from declarative definitions of the grammar, plus actions to take when grammatical constructs are parsed successfully.
- The handout and tarballs are available on CATE and at: `http://www.doc.ic.ac.uk/~dcw/c-tools-2018/lecture4/`

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex,

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.
  - In 03.expr3 we add named constants into the grammar, using our longhash module.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.
  - In 03.expr3 we add named constants into the grammar, using our longhash module.
  - In 04.expr4 we use our macro processor to make the evaluation actions easier.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.
  - In 03.expr3 we add named constants into the grammar, using our longhash module.
  - In 04.expr4 we use our macro processor to make the evaluation actions easier.
  - Finally, in 05.expr5 change the evaluation actions to tree-building actions using datadec to define the tree types, and then evaluate the expression by walking the tree.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.
  - In 03.expr3 we add named constants into the grammar, using our longhash module.
  - In 04.expr4 we use our macro processor to make the evaluation actions easier.
  - Finally, in 05.expr5 change the evaluation actions to tree-building actions using datadec to define the tree types, and then evaluate the expression by walking the tree.
- I used to present several of those examples here, over half a dozen slides, taking half the lecture.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.
  - In 03.expr3 we add named constants into the grammar, using our longhash module.
  - In 04.expr4 we use our macro processor to make the evaluation actions easier.
  - Finally, in 05.expr5 change the evaluation actions to tree-building actions using datadec to define the tree types, and then evaluate the expression by walking the tree.
- I used to present several of those examples here, over half a dozen slides, taking half the lecture. But this year, I've decided to try something different.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.
  - In 03.expr3 we add named constants into the grammar, using our longhash module.
  - In 04.expr4 we use our macro processor to make the evaluation actions easier.
  - Finally, in 05.expr5 change the evaluation actions to tree-building actions using datadec to define the tree types, and then evaluate the expression by walking the tree.
- I used to present several of those examples here, over half a dozen slides, taking half the lecture. But this year, I've decided to try something different.
- I'm going to present only one example of using Lex and Yacc: a complex one.

- In the tarball, you will find a whole set of worked examples of using Yacc and Lex, choosing as our little language integer constant expressions such as

  `3*(10+16*(123/3) mod 7)`

  - In 01.expr1 we start with a plain grammar and lexer.
  - In 02.expr2 we add evaluation actions to build a calculator.
  - In 03.expr3 we add named constants into the grammar, using our longhash module.
  - In 04.expr4 we use our macro processor to make the evaluation actions easier.
  - Finally, in 05.expr5 change the evaluation actions to tree-building actions using datadec to define the tree types, and then evaluate the expression by walking the tree.
- I used to present several of those examples here, over half a dozen slides, taking half the lecture. But this year, I've decided to try something different.
- I'm going to present only one example of using Lex and Yacc: a complex one. So this is an experiment - let's see what happens:-)

- Let's define a tiny Haskell subset called THS.

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc.

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.
- Ok, what Haskell subset should we choose?
  - Zero-or-more function definitions, with optional type definitions,

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.
- Ok, what Haskell subset should we choose?
  - Zero-or-more function definitions, with optional type definitions,
  - Followed by a compulsory integer expression (often a call to some of those functions).

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.
- Ok, what Haskell subset should we choose?
  - 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,

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.
- Ok, what Haskell subset should we choose?
  - 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 implemented either by a single expression, or

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.
- Ok, what Haskell subset should we choose?
  - 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 implemented either by a single expression, or
  - A sequence of guarded expressions involving simple boolean expressions, eg. `x==0`,

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.
- Ok, what Haskell subset should we choose?
  - 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 implemented either by a single expression, or
  - A sequence of guarded expressions involving simple boolean expressions, eg. `x==0`,
- For example:

```
f :: Int -> Int
f 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)
f(20) + abs(0-2)*fact(arg1)
```

- Let's define a tiny Haskell subset called THS. Then build a Lexer and Parser using Lex and Yacc. Then build an Abstract Syntax Tree using Datadec and Yacc tree-building actions.
- Ok, what Haskell subset should we choose?
  - 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 implemented either by a single expression, or
  - A sequence of guarded expressions involving simple boolean expressions, eg. `x==0`,
- For example:

```
f :: Int -> Int
f 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)
f(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:-)

- The basic lexical tokens we need are:
  - A few keywords ‘mod’, ‘Int’, ‘True’.
  - Various one-or-two character tokens (eg. ‘(’, ‘+’, ‘*’, ‘)’, ‘::’ etc).
  - Numeric constants (eg ‘123’).
  - Identifiers (eg ‘fact’).

- The basic lexical tokens we need are:
  - A few keywords 'mod', 'Int', 'True'.
  - Various one-or-two character tokens (eg. '(', '+', '*', ')', '::' etc).
  - Numeric constants (eg '123').
  - Identifiers (eg 'fact').
- With Lex, specify the tokens as regular expression/action pairs:

```
[ \t\n]+                /* ignore whitespace */;
mod                     return MOD;
Int                     return INTTYPE;
True                    return TRUEV;
::                      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;
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext);return IDENT;
.                       return TOKERR;
```

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

- Note that we are being extremely minimal with our tokens, including (for example) True but not False, '>' but not '<' etc. These can trivially be added later.
- See lexer.l for the full Lex input file, containing the above plus some prelude. This file can be turned into compilable C code via: lex -o lexer.c lexer.l.

- Note that we are being extremely minimal with our tokens, including (for example) True but not False, '>' but not '<' etc. These can trivially be added later.
- See lexer.l for the full Lex input file, containing the above plus some prelude. This file can be turned into compilable C code via: lex -o lexer.c lexer.l.
- Our next task is to combine these tokens into THS programs via our grammar. However the grammar and (Datadec-generated) Abstract Syntax Trees intertwine, so 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 );
```

- Now let's look at the Yacc input file parser.y, it starts with a long prelude of plain C code:

```
%{
// some includes

extern int yylex (void);
extern int yylineno;
extern bool verbose;

program ast  = NULL;
int yyerrors = 0;

void yyerror(const char *str)
{
        fprintf(stderr,"line %d: error: %s\n", yylineno, str);
        yyerrors++;
}

int yywrap( void ) { return 1; }
%}
```

- Note that among the prelude, we see:

```
program ast = NULL;
```

which is where the AST (the program) will be stored after a successful parse.

- Next parser.y contains a %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;
}
```

In the generated C code, the union is turned into a type called YYSTYPE in parser.h. The Lex prelude includes parser.h, and Lex then defines the variable YYSTYPE yylval, which explains how yylval.n is an int, and yylval.s is a char *.

- Next parser.y contains a %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;
}
```

  In the generated C code, the union is turned into a type called YYSTYPE in parser.h. The Lex prelude includes parser.h, and Lex then defines the variable YYSTYPE yylval, which explains how yylval.n is an int, and yylval.s is a char *.
- Next, parser.y defines all the tokens:

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

  Yacc turns each token into an integer constant which the lexer uses (via including parser.h). The final two lines tell Yacc that a NUMBER token has an associated integer value (int n in the union), and that an IDENT token has an associated char *s (identifier name).

- Next, we do the same for those grammar rules with associated data: we associate a specific field in the union with particular rules:

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

- Next, we tell Yacc which rule to start parsing with:

```
%start program
%%
```

- Then we list the grammar rules and corresponding tree-building actions to take:

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

- I'll explain all the strange `$n` and `$$` syntax shortly.

- The grammar rules continue, defining guarded expressions (guard), boolean expressions (bexpr) and arithmetic expressions (rules expr, term and factor).
- Picking one rule out, we see:

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

  This means that one possible way of parsing a guarded expression is to find a GUARD token (the '|' symbol), followed immediately by an arbitrarily complicated boolean expression, followed by an IS token (the '=' symbol), followed by an expression.

- The grammar rules continue, defining guarded expressions (guard), boolean expressions (bexpr) and arithmetic expressions (rules expr, term and factor).
- Picking one rule out, we see:

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

  This means that one possible way of parsing a guarded expression is to find a GUARD token (the '|' symbol), followed immediately by an arbitrarily complicated boolean expression, followed by an IS token (the '=' symbol), followed by an expression.
- If this rule matches, then the action is executed, with:
  - $1 set to the value (if any) associated with the GUARD token,
  - $2 set to the value (if any) associated with the bexpr rule,
  - $3 set to the value (if any) associated with the IS token, and
  - $4 set to the value (if any) associated with the expr rule.

- The grammar rules continue, defining guarded expressions (guard), boolean expressions (bexpr) and arithmetic expressions (rules expr, term and factor).
- Picking one rule out, we see:

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

  This means that one possible way of parsing a guarded expression is to find a GUARD token (the '|' symbol), followed immediately by an arbitrarily complicated boolean expression, followed by an IS token (the '=' symbol), followed by an expression.
- If this rule matches, then the action is executed, with:
  - $1 set to the value (if any) associated with the GUARD token,
  - $2 set to the value (if any) associated with the bexpr rule,
  - $3 set to the value (if any) associated with the IS token, and
  - $4 set to the value (if any) associated with the expr rule.
- Here, only the bexpr and the expr have associated values, so we use $2 and $4 to build a guard: `guard_pair( $2, $4 )`.

- The grammar rules continue, defining guarded expressions (guard), boolean expressions (bexpr) and arithmetic expressions (rules expr, term and factor).
- Picking one rule out, we see:

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

  This means that one possible way of parsing a guarded expression is to find a GUARD token (the '|' symbol), followed immediately by an arbitrarily complicated boolean expression, followed by an IS token (the '=' symbol), followed by an expression.
- If this rule matches, then the action is executed, with:
  - $1 set to the value (if any) associated with the GUARD token,
  - $2 set to the value (if any) associated with the bexpr rule,
  - $3 set to the value (if any) associated with the IS token, and
  - $4 set to the value (if any) associated with the expr rule.
- Here, only the bexpr and the expr have associated values, so we use $2 and $4 to build a guard: `guard_pair( $2, $4 )`.
- Assigning that new guard to $$ sets the value associated with the whole guard rule, think of this as the return value.

- Note that recursive (list-handling) rules in Yacc, such as:

  ```
  guardrules : guardrules guard  { ACTION }
  ```

  must be written with the recursive invocation first. If we write the action as `$$ = guardlist_cons($2,$1)` we would generate the list in reverse order.

- Note that recursive (list-handling) rules in Yacc, such as:

```
guardrules : guardrules guard  { ACTION }
```

  must be written with the recursive invocation first. If we write the action as `$$ = guardlist_cons($2,$1)` we would generate the list in reverse order.
- Instead, the action is `$$ = guardlist_push($1,$2)`. This function was manually written (you'll find it in types.in) and modifies the existing guardlist, finding the last node and adding the new guard there. That's fine when we're building the list up.

- Note that recursive (list-handling) rules in Yacc, such as:

```
guardrules : guardrules guard  { ACTION }
```

  must be written with the recursive invocation first. If we write the action as `$$ = guardlist_cons($2,$1)` we would generate the list in reverse order.
- Instead, the action is `$$ = guardlist_push($1,$2)`. This function was manually written (you'll find it in types.in) and modifies the existing guardlist, finding the last node and adding the new guard there. That's fine when we're building the list up.
- Turn parser.y into C code (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.

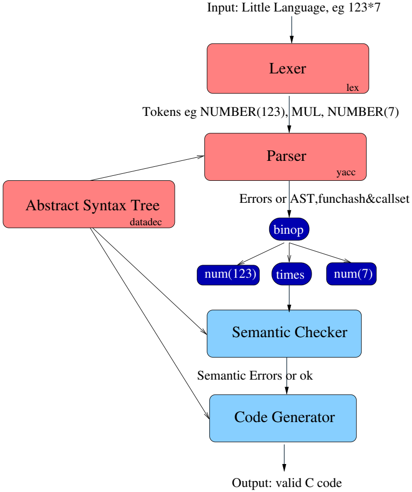
- Note that recursive (list-handling) rules in Yacc, such as:

```
guardrules : guardrules guard  { ACTION }
```

  must be written with the recursive invocation first. If we write the action as `$$ = guardlist_cons($2,$1)` we would generate the list in reverse order.
- Instead, the action is `$$ = guardlist_push($1,$2)`. This function was manually written (you'll find it in types.in) and modifies the existing guardlist, finding the last node and adding the new guard there. That's fine when we're building the list up.
- Turn parser.y into C code (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.
- Putting it all together, using our macro tool from the previous lecture, and adding a main program that initializes the lexer, invokes the parser and (when parsing is successful) prints out the AST that was built, plus several other modules we haven't discussed, and a Makefile, compile and link by typing make.
- We end up with a THS parser and treebuilder ths1, in which we only write about 460 lines of code. Give it a try!

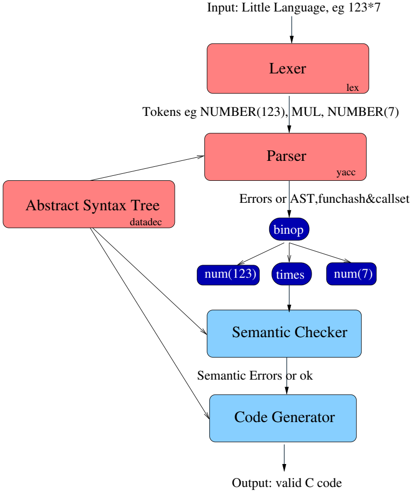
- 07.ths-codegen extends our treebuilder, adding semantic checking (eg. checking that we define every function we call) and then code generation - translating THS to C!
- How do we do semantic checks? A semantic checker either walks the AST, or builds and iterates over equivalent data structures.
- In fact, to reduce tree-walking, we enhanced parser.y to create a hash and a set as well: the funchash maps from a defined functionname to it's AST representation. The callset names all called functions.
- The Semantic checker then iterates through the callset checking that each called function is present in the funchash.
- 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 130 lines) was custom C code, mainly building and sorting an array of functions, then invoking datadec-generated print_TYPE() functions.

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



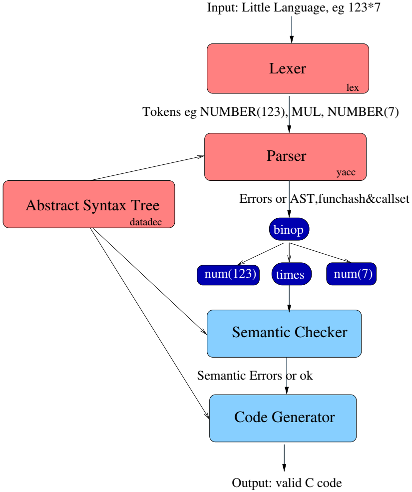
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.

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



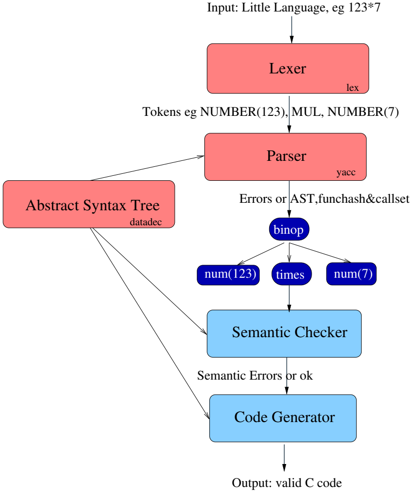
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar,

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



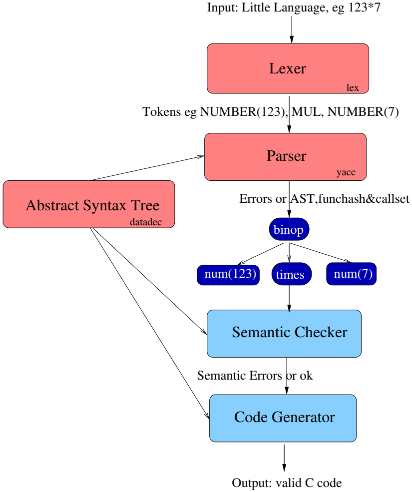
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar, builds an AST

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



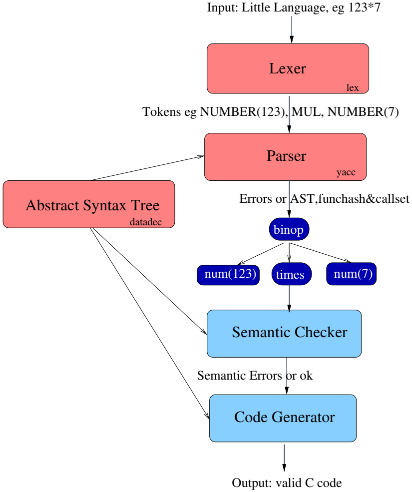
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar, builds an AST and builds funchash and callset.

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



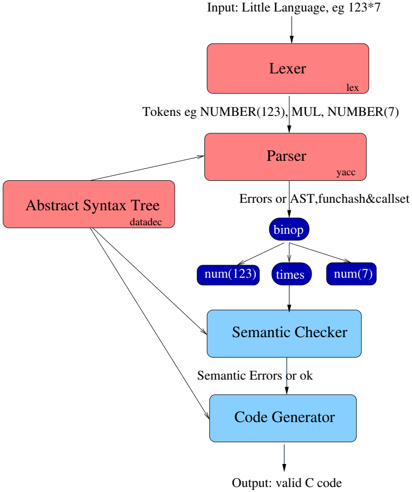
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar, builds an AST and builds funchash and callset.
- Our Semantic checker uses the funchash and callset

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.
- Our Semantic checker uses the funchash and callset to check that there are no consistency problems.

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.
- Our Semantic checker uses the 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.

- 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 850 lines of code ourselves.

- 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 850 lines of code ourselves.
- However, after datadec, macro, Yacc and Lex have run, there are approximately 5100 lines of C code (including headers) overall.

- 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 850 lines of code ourselves.
- However, after datadec, macro, Yacc and Lex have run, there are approximately 5100 lines of C code (including headers) overall.
- 850/5100 is about 16%.

- 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 850 lines of code ourselves.
- However, after datadec, macro, Yacc and Lex have run, there are approximately 5100 lines of C code (including headers) overall.
- 850/5100 is about 16%.
- To put that another way: *our tools wrote 84% of the code for us*.

- 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 850 lines of code ourselves.
- However, after datadec, macro, Yacc and Lex have run, there are approximately 5100 lines of C code (including headers) overall.
- 850/5100 is about 16%.
- To put that another way: *our tools wrote 84% of the code for us*.
- That's pretty impressive - very few combinations of tools automate anywhere near that much of our 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 07.ths-codegen, we have only written about 850 lines of code ourselves.
- However, after datadec, macro, Yacc and Lex have run, there are approximately 5100 lines of C code (including headers) overall.
- 850/5100 is about 16%.
- To put that another way: *our tools wrote 84% 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.

- 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 850 lines of code ourselves.
- However, after datadec, macro, Yacc and Lex have run, there are approximately 5100 lines of C code (including headers) overall.
- 850/5100 is about 16%.
- To put that another way: *our tools wrote 84% 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 order to completely make sense of how they all fit together, with the `%union` and the `%type <f>` and `%token <f>` syntax and all the `$n` notation, please work slowly through the much smaller examples of Yacc and Lex from the tarball (parsing and manipulating expressions).

- More recently, I've been playing with an entirely different approach to language parsing:
- Suppose instead of defining a complete little language, we want to add a single well-defined feature to a large language like (say) C. We work out the syntax of the new feature, and define it's semantics via a precise description of how to translate it back to standard C.

- More recently, I've been playing with an entirely different approach to language parsing:
- Suppose instead of defining a complete little language, we want to add a single well-defined feature to a large language like (say) C. We work out the syntax of the new feature, and define it's semantics via a precise description of how to translate it back to standard C.
- We could get a complete C compiler and graft our new feature into it. That might be simple, or a nightmare.
- Or we could get a complete Yacc/Lex C grammar and extend that - adding our new feature. That's relatively easy.

- More recently, I've been playing with an entirely different approach to language parsing:
- Suppose instead of defining a complete little language, we want to add a single well-defined feature to a large language like (say) C. We work out the syntax of the new feature, and define it's semantics via a precise description of how to translate it back to standard C.
- We could get a complete C compiler and graft our new feature into it. That might be simple, or a nightmare.
- Or we could get a complete Yacc/Lex C grammar and extend that - adding our new feature. That's relatively easy.
- But we can't ignore all the boring standard C. We have to deal with it somehow. We could build and walk a complete AST (turning our new feature back into standard C, while re-generating all the unaltered C code).

- More recently, I've been playing with an entirely different approach to language parsing:
- Suppose instead of defining a complete little language, we want to add a single well-defined feature to a large language like (say) C. We work out the syntax of the new feature, and define it's semantics via a precise description of how to translate it back to standard C.
- We could get a complete C compiler and graft our new feature into it. That might be simple, or a nightmare.
- Or we could get a complete Yacc/Lex C grammar and extend that - adding our new feature. That's relatively easy.
- But we can't ignore all the boring standard C. We have to deal with it somehow. We could build and walk a complete AST (turning our new feature back into standard C, while re-generating all the unaltered C code).
- But there's a lot of "do-nothing" work going on here.
- Is there any way of avoiding this?

- Yes! graft our new feature into C by writing a simple line-by-line pre-processor that copies most lines through unchanged (hoping they're valid C), but locates specially marked extension directives, turning each into a corresponding chunk of plain C.

- Yes! graft our new feature into C by writing a simple line-by-line pre-processor that copies most lines through unchanged (hoping they're valid C), but locates specially marked extension directives, turning each into a corresponding chunk of plain C.
- In 08.cm-translator you'll find a Perl script called CM that grafts a simple "C with Modules" syntax onto the front of C. An example tiny.cm CM input file:

```
//  tiny "C+Module" example: a 100 element array type and one operation on it
#include <stdio.h>
#include <stdbool.h>
%pubconst MAXELEMENTS 100
%defn
{
        typedef int array[MAXELEMENTS];
}
%pubfunc bool ok = initialize( array x )
        initialize an array, did it work?
{
        for( int i=0; i<MAXELEMENTS; i++ )
        {
                x[i] = 0;
        }
        return true;
}
```

- CM turns this into a complete plain C module - tiny.c and tiny.h. See intstack.cm for a bigger example.

- Follow 100,000 years of human history by tool-using and tool-making.

- Follow 100,000 years of human history by tool-using and tool-making. Build yourself a powerful toolkit.

- 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.

- 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!

- 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.

- 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.

- 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/`

- 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/`

- 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/`
- Most importantly: enjoy your C programming! Build your toolkit - and let me know if you find, or build, any particularly cool tools!

- 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/`
- Most importantly: enjoy your C programming! Build your toolkit - and let me know if you find, or build, any particularly cool tools!
- Finally, read The Pragmatic Programmer.

- 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/`
- Most importantly: enjoy your C programming! Build your toolkit - and let me know if you find, or build, any particularly cool tools!
- Finally, read The Pragmatic Programmer. That's all folks!