C Programming Tools: Part 4

Building Lexers and Parsers

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

Dept of Computing,
Imperial College London

June 2019

The handout and tarball are available on CATE and at:
`http://www.doc.ic.ac.uk/~dcw/c-tools-2019/lecture4/`

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

- Last lecture, 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 lecture, 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 lecture, 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.
- Now, in the last C Programming Tools lecture, we'll find how to make writing Code Generators even easier.

- Last lecture, 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.
- Now, 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 them, but..

- Last lecture, 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.
- Now, 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 them, but..
- This problem has been solved!

- Last lecture, 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.
- Now, 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 them, but..
- This problem has been solved! Lex generates C code (a Lexer) from declarative definitions of lexical tokens.

- Last lecture, 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.
- Now, 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 them, 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 parser calls the lexer to supply the next token.

- Last lecture, 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.
- Now, 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 them, 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 parser calls the lexer to supply the next token.
- But what Little Language shall we use for our main example?

- A tiny Haskell subset called THS.

- A tiny Haskell subset called THS. We'll build a Lexer and Parser using Lex and Yacc.

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

- A tiny Haskell subset called THS. We'll 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,

- A tiny Haskell subset called THS. We'll 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).

- A tiny Haskell subset called THS. We'll 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,

- A tiny Haskell subset called THS. We'll 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 tiny Haskell subset called THS. We'll 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`.

- A tiny Haskell subset called THS. We'll 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)
```

- A tiny Haskell subset called THS. We'll 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 ‘2’ or ‘123’).
  - Identifiers (eg ‘fact’ or ‘x’).

- The basic lexical tokens we need are:
  - A few keywords 'mod', 'Int', 'True'.
  - Various one-or-two character tokens (eg. '(', '+', '*', ')', '::' etc).
  - Numeric constants (eg '2' or '123').
  - Identifiers (eg 'fact' or 'x').
- 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]+                  return NUMBER;
[a-z][a-z0-9]*          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 in 01.ths-recogniser 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.

- 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 in 01.ths-recogniser 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.
- Our next task is to combine these tokens into THS programs via our grammar. The Yacc input file parser.y starts with a long prelude of plain C code:

```
%{
// some includes

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

int yyerrors = 0;

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

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

- 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 NUMBER IDENT
```

- 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 NUMBER IDENT
```

- Yacc turns each token into an integer constant which the lexer uses (because lexer.l's preamble includes parser.h). This is how the generated lexer can use those token values in actions.

- 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 NUMBER IDENT
```

- Yacc turns each token into an integer constant which the lexer uses (because lexer.l's preamble includes parser.h). This is how the generated lexer can use those token values in actions.
- Next, parser.y tells Yacc which rule to start parsing with:

```
%start program
%%
```

  The generated parser will try to consume the entire input and parse it as a program.

- 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 NUMBER IDENT
```

- Yacc turns each token into an integer constant which the lexer uses (because lexer.l's preamble includes parser.h). This is how the generated lexer can use those token values in actions.
- Next, parser.y tells Yacc which rule to start parsing with:

```
%start program
%%
```

  The generated parser will try to consume the entire input and parse it as a program.
- Then we list the grammar rules that define the language, in BNF:

```
program      :  defns expr
             ;
defns        :  /* empty */
             | defns ftypedefn
             | defns fdefinition
             ;
ftypedefn    : IDENT COLONCOLON INTTYPE IMPLIES INTTYPE
             ;
fdefinition  : IDENT IDENT IS expr
             | IDENT IDENT guardrules
             ;
guardrules   : guard
             | guardrules guard
             ;
```

- The grammar rules continue, defining guarded expressions (guard), boolean expressions (bexpr) and arithmetic expressions (rules expr, term and factor):

```
guard       : GUARD bexpr IS expr
            ;
bexpr       : expr EQ expr
            | expr NE expr
            | expr GT expr
            | TRUEV
            ;
expr        : expr PLUS term
            | expr MINUS term
            | term
            ;
term        : term MUL factor
            | term DIV factor
            | term MOD factor
            | factor
            ;
factor      : NUMBER
            | OPEN expr CLOSE
            | IDENT OPEN expr CLOSE
            | IDENT
            ;
```

- The grammar rules continue, defining guarded expressions (guard), boolean expressions (bexpr) and arithmetic expressions (rules expr, term and factor):

```
guard       : GUARD bexpr IS expr
            ;
bexpr       : expr EQ expr
            | expr NE expr
            | expr GT expr
            | TRUEV
            ;
expr        : expr PLUS term
            | expr MINUS term
            | term
            ;
term        : term MUL factor
            | term DIV factor
            | term MOD factor
            | factor
            ;
factor      : NUMBER
            | OPEN expr CLOSE
            | IDENT OPEN expr CLOSE
            | IDENT
            ;
```

- Picking the top guard rule out as an example, it means that a guarded expression comprises a GUARD token (‘|’), followed by a boolean expression, followed by an IS token (‘=’), followed by an expression.

- Note that recursive rules in Yacc that match lists of items, eg:

```
defns      : defns ftypedefn
defns      : defns fdefinition
guardrules : guardrules guard
```

  must be written with the recursive invocation first.

- Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you do. We'll see a complication that this causes later.

- Note that recursive rules in Yacc that match lists of items, eg:

```
defns      : defns ftypedefn
defns      : defns fdefinition
guardrules : guardrules guard
```

  must be written with the recursive invocation first.
- Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you do. We'll see a complication that this causes later.
- Turn parser.y into a C module (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.

- Note that recursive rules in Yacc that match lists of items, eg:

```
defns      : defns ftypedefn
defns      : defns fdefinition
guardrules : guardrules guard
```

  must be written with the recursive invocation first.
- Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you do. We'll see a complication that this causes later.
- Turn parser.y into a C module (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.
- Putting it all together, adding a main program that initializes the lexer, invokes the parser, and checks whether parsing is successful or not, and adding a Makefile, compile and link by typing make.
- We end up with a THS recogniser ths1, in which we only write about 170 lines of code. Give it a try!

- Note that recursive rules in Yacc that match lists of items, eg:

```
defns      : defns ftypedefn
defns      : defns fdefinition
guardrules : guardrules guard
```

  must be written with the recursive invocation first.
- Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you do. We'll see a complication that this causes later.
- Turn parser.y into a C module (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.
- Putting it all together, adding a main program that initializes the lexer, invokes the parser, and checks whether parsing is successful or not, and adding a Makefile, compile and link by typing make.
- We end up with a THS recogniser ths1, in which we only write about 170 lines of code. Give it a try!
- Our next THS version, in 02.ths-treebuilder, will build an Abstract Syntax tree during the parse!

- First, we must alter our lexer.l slightly: when the lexer recognises a number, it's not enough to say *"it's a number"* - which number is it? Ditto for an identifier.

- First, we must alter our lexer.l slightly: when the lexer recognises a number, it's not enough to say *"it's a number"* - which number is it? Ditto for an identifier.
- To do this, we make two tiny changes to lexer.l:

```
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext); return IDENT;
```

- First, we must alter our lexer.l slightly: when the lexer recognises a number, it's not enough to say *"it's a number"* - which number is it? Ditto for an identifier.
- To do this, we make two tiny changes to lexer.l:

```
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext); return IDENT;
```

- When a Lex pattern matches a chunk of input, the input is stored by the lexer in a variable called char yytext[] before the action runs.

- First, we must alter our lexer.l slightly: when the lexer recognises a number, it's not enough to say *"it's a number"* - which number is it? Ditto for an identifier.
- To do this, we make two tiny changes to lexer.l:

```
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext); return IDENT;
```

- When a Lex pattern matches a chunk of input, the input is stored by the lexer in a variable called char yytext[] before the action runs.
- So, when the pattern [0-9]+ has matched, the longest possible digit sequence at the head of the unconsumed input is stored in yytext.

- First, we must alter our lexer.l slightly: when the lexer recognises a number, it's not enough to say *"it's a number"* - which number is it? Ditto for an identifier.
- To do this, we make two tiny changes to lexer.l:

```
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext); return IDENT;
```

- When a Lex pattern matches a chunk of input, the input is stored by the lexer in a variable called char yytext[] before the action runs.
- So, when the pattern [0-9]+ has matched, the longest possible digit sequence at the head of the unconsumed input is stored in yytext.
- Then our action extracts the integer value from yytext via atoi(yytext) and stores it in the mysterious yylval.n. Then it returns NUMBER.
- What is yylval.n? We'll answer that on the next slide.

- First, we must alter our lexer.l slightly: when the lexer recognises a number, it's not enough to say *"it's a number"* - which number is it? Ditto for an identifier.
- To do this, we make two tiny changes to lexer.l:

```
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext); return IDENT;
```

- When a Lex pattern matches a chunk of input, the input is stored by the lexer in a variable called char yytext[] before the action runs.
- So, when the pattern [0-9]+ has matched, the longest possible digit sequence at the head of the unconsumed input is stored in yytext.
- Then our action extracts the integer value from yytext via atoi(yytext) and stores it in the mysterious yylval.n. Then it returns NUMBER.
- What is yylval.n? We'll answer that on the next slide.
- Similarly, when we've matched an identifier - the longest possible alphanumeric sequence that isn't a keyword such as "mod" - the name of the identifier is in yytext. We strdup(yytext) to give ourselves a long-lived copy of the string, and store that in yylval.s - a char *s field in the mysterious yylval.

- Moving onto parser.y, there are many changes. First, the parser.y prelude includes rather more `#include` statements, then defines:

```
program ast = NULL;
```

  which is where the AST (the program) is stored after a successful parse. We'll see where the type program is defined shortly.

- Moving onto parser.y, there are many changes. First, the parser.y prelude includes rather more `#include` statements, then defines:

```
program ast = NULL;
```

  which is where the AST (the program) is stored after a successful parse. We'll see where the type program is defined shortly.

- 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;
}
```

- Moving onto parser.y, there are many changes. First, the parser.y prelude includes rather more `#include` statements, then defines:

```
program ast = NULL;
```

  which is where the AST (the program) is stored after a successful parse. We'll see where the type program is defined shortly.

- 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 union type called YYSTYPE in parser.h, which declares extern YYSTYPE yylval, and parser.c defines the variable YYSTYPE yylval.

- Moving onto parser.y, there are many changes. First, the parser.y prelude includes rather more `#include` statements, then defines:

```
program ast = NULL;
```

  which is where the AST (the program) is stored after a successful parse. We'll see where the type program is defined shortly.

- 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 union type called YYSTYPE in parser.h, which declares extern YYSTYPE yylval, and parser.c defines the variable YYSTYPE yylval.
- The Lex prelude includes parser.h: this explains how yylval.n is an int, and yylval.s is a char *.

- Next, parser.y alters the definition of these two tokens:

```
%token <n> NUMBER
%token <s> IDENT
```

This tells Yacc that a NUMBER token has an associated int n value, and that an IDENT token has an associated char *s value.

- Next, parser.y alters the definition of these two tokens:

```
%token <n> NUMBER
%token <s> IDENT
```

  This tells Yacc that a NUMBER token has an associated int n value, and that an IDENT token has an associated char *s value.

- But the %union contained other fields - of types such as expr, bexpr, guard, guardlist, fdefn and flist - plus we have already seen the type program.

- Next, parser.y alters the definition of these two tokens:

```
%token <n> NUMBER
%token <s> IDENT
```

  This tells Yacc that a NUMBER token has an associated int n value, and that an IDENT token has an associated char *s value.

- But the %union contained other fields - of types such as expr, bexpr, guard, guardlist, fdefn and flist - plus we have already seen the type program.
- These AST types are defined in types.in - a *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     =  triple( string fname, string param, fbody b );
fbody     =  one( expr e ) or many( guardlist l );
flist     =  nil or cons( fdefn hd, flist tl );
program   =  pair( flist l, expr e );
```

- Back in parser.y, we associate a specific field in the union with many, but not all, grammar rules:

```
%type <e> factor term expr
%type <g> guard
%type <f> fdefinition
```

```
%type <b> bexpr
%type <gl> guardrules
%type <fl> defns
```

- Back in parser.y, we associate a specific field in the union with many, but not all, grammar rules:

```
%type <e> factor term expr
%type <g> guard
%type <f> fdefinition
```

```
%type <b> bexpr
%type <gl> guardrules
%type <fl> defns
```

- You will see that all the grammar rules have been annotated with the corresponding tree-building actions to take when the rules match:

```
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 the strange $n and $$ syntax shortly.

- Picking one of our rules + actions out, we see:

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

- Picking one of our rules + actions out, we see:

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

- 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.

- Picking one of our rules + actions out, we see:

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

- 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 )`.

- Picking one of our rules + actions out, we see:

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

- 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.

- Picking one of our rules + actions out, we see:

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

- 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.
- Having built a new guard, parsing continues trying to parse a non-empty sequence of guards, and build them into a guard list. Guard lists get incorporated into function definitions, function definitions into a function list, the function list and the main expression into the program - and assigned to the ast variable.

- We said that recursive 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.

- We said that recursive 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.

- We said that recursive 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.
- We modify the main program to print out the AST (if parsing is successful), add named constants (via the longhash module), and modify the Makefile to build everything, using **lex, yacc** and **datadec** to generate lexer.c, the parser module and the types module, using the **tmpl** and **proto** tools from the previous lecture to generate numerous small modules.

- We said that recursive 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.
- We modify the main program to print out the AST (if parsing is successful), add named constants (via the longhash module), and modify the Makefile to build everything, using **lex, yacc** and **datadec** to generate lexer.c, the parser module and the types module, using the **tmpl** and **proto** tools from the previous lecture to generate numerous small modules.
- Compile and link by typing make. We end up with a THS parser and treebuilder ths2, in which we only write about 460 lines of code. Give it a try!

- 03.ths-semanticchecker adds semantic checking - in THS, this means checking that we define every function we call. In other languages, we'd have to check 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 to populate a hash called funchash as we parse functions, and a set called callset as we parse calls.
  - The funchash maps from a defined functionname to it's AST representation (a fdefn).
  - The callset is a set of all functions that are called.

- 03.ths-semanticchecker adds semantic checking - in THS, this means checking that we define every function we call. In other languages, we'd have to check 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 to populate a hash called funchash as we parse functions, and a set called callset as we parse calls.
  - The funchash maps from a defined functionname to it's AST representation (a fdefn).
  - The callset is a set of all functions that are called.
- We also add a function called check_id() to parser.y that expands named constants into numbers as early as possible, building an expr_num(n) instead of the usual expr_id(name).

- 03.ths-semanticchecker adds semantic checking - in THS, this means checking that we define every function we call. In other languages, we'd have to check 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 to populate a hash called funchash as we parse functions, and a set called callset as we parse calls.
  - The funchash maps from a defined functionname to it's AST representation (a fdefn).
  - The callset is a set of all functions that are called.
- We also add a function called check_id() to parser.y that expands named constants into numbers as early as possible, building an expr_num(n) instead of the usual expr_id(name).
- The semantic checker then iterates through the callset checking that each called function is present in the funchash.

- 04.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.

- 04.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 must construct C functions to:
  - Evaluate an integer expression in the current environment.
  - Evaluate a boolean expression in the current environment.
  - To select which guard in a guardlist is true and then evaluate it's corresponding integer expression, all in the current environment.
  - To handle a function call (possibly recursive).

- 04.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 must construct C functions to:
  - Evaluate an integer expression in the current environment.
  - Evaluate a boolean expression in the current environment.
  - To select which guard in a guardlist is true and then evaluate it's corresponding integer expression, all in the current environment.
  - To handle a function call (possibly recursive).
- The only tricky part is that in a function call, we evaluate the actual parameter expression down to an integer in the current environment, and then evaluate the function body (either an expression or a guardlist) with a new environment in which the function's parameter variable is set to that integer value.

- 04.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 must construct C functions to:
  - Evaluate an integer expression in the current environment.
  - Evaluate a boolean expression in the current environment.
  - To select which guard in a guardlist is true and then evaluate it's corresponding integer expression, all in the current environment.
  - To handle a function call (possibly recursive).
- The only tricky part is that in a function call, we evaluate the actual parameter expression down to an integer in the current environment, and then evaluate the function body (either an expression or a guardlist) with a new environment in which the function's parameter variable is set to that integer value.
- If we do this right, our interpreter will correctly handle recursion.

- 04.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 must construct C functions to:
  - Evaluate an integer expression in the current environment.
  - Evaluate a boolean expression in the current environment.
  - To select which guard in a guardlist is true and then evaluate it's corresponding integer expression, all in the current environment.
  - To handle a function call (possibly recursive).
- The only tricky part is that in a function call, we evaluate the actual parameter expression down to an integer in the current environment, and then evaluate the function body (either an expression or a guardlist) with a new environment in which the function's parameter variable is set to that integer value.
- 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, 05.ths-codegen, replaces our interpreter with a code generator - which translates THS to C!

- Our final version of THS, 05.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!

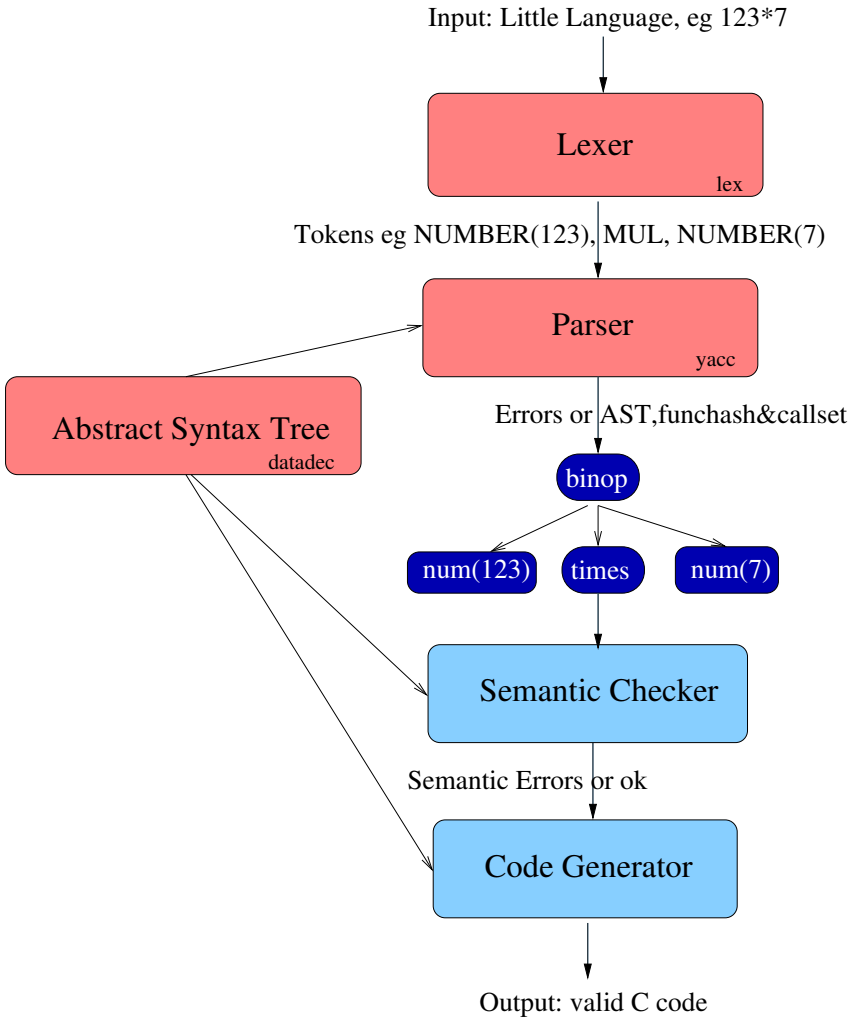
- Our final version of THS, 05.ths-codegen, replaces our interpreter with a code generator - which translates THS to C!
- How do we do code generation? A code generator is just another AST and funchash walker, one with suitable print statements!
- In fact, using datadec's print hints mechanism, 80% of the C code generation was done by making each AST type print itself in valid C form.

- Our final version of THS, 05.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 130 lines) was custom C code, gluing everything together.

- Our final version of THS, 05.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 130 lines) was custom C code, gluing everything together.
- One subtlety was that Haskell (and 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. I choose to generate these prototypes in alphabetically sorted order - so I built and sort an array of fdefns before printing out prototypes using the array.

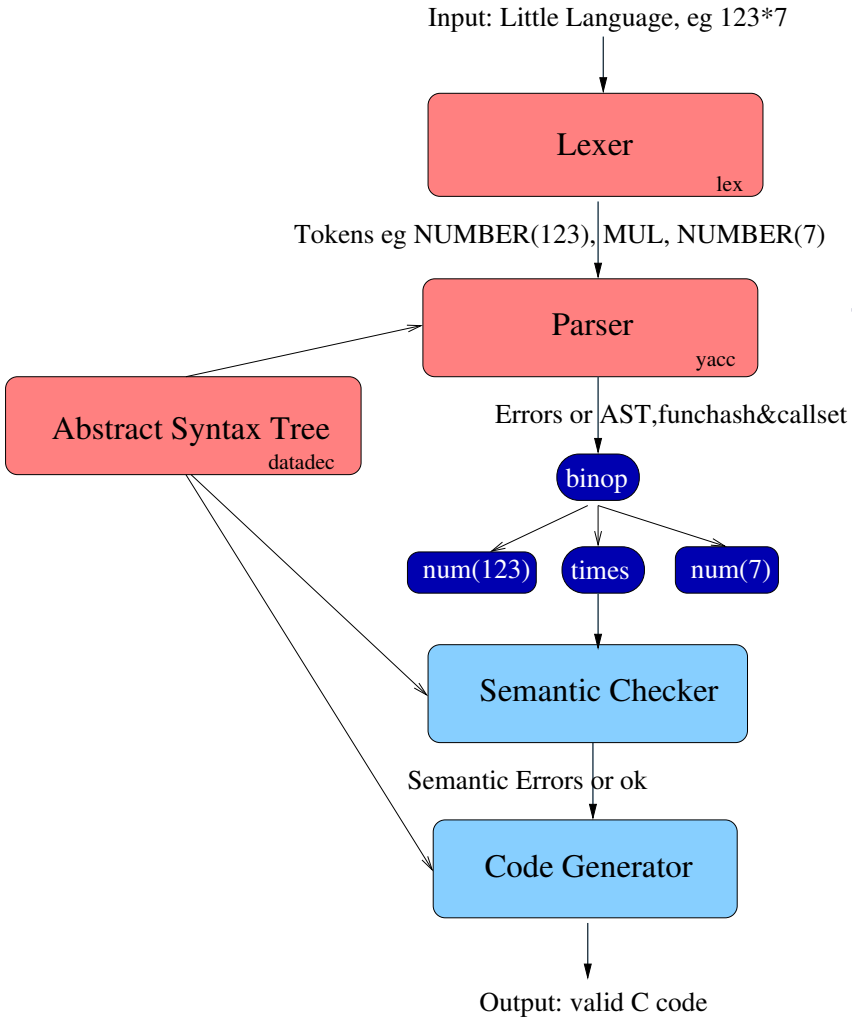
- Our final version of THS, 05.ths-codegen, replaces our interpreter with a code generator - which translates THS to C!
- How do we do code generation? A code generator is just another AST and funchash walker, one with suitable print statements!
- In fact, using datadec's print hints mechanism, 80% of the C code generation was done by making each AST type print itself in valid C form.
- The remaining 20% (approx 130 lines) was custom C code, gluing everything together.
- One subtlety was that Haskell (and 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. I choose to generate these prototypes in alphabetically sorted order - so I built and sort an array of fdefns before printing out prototypes using the array.
- 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:



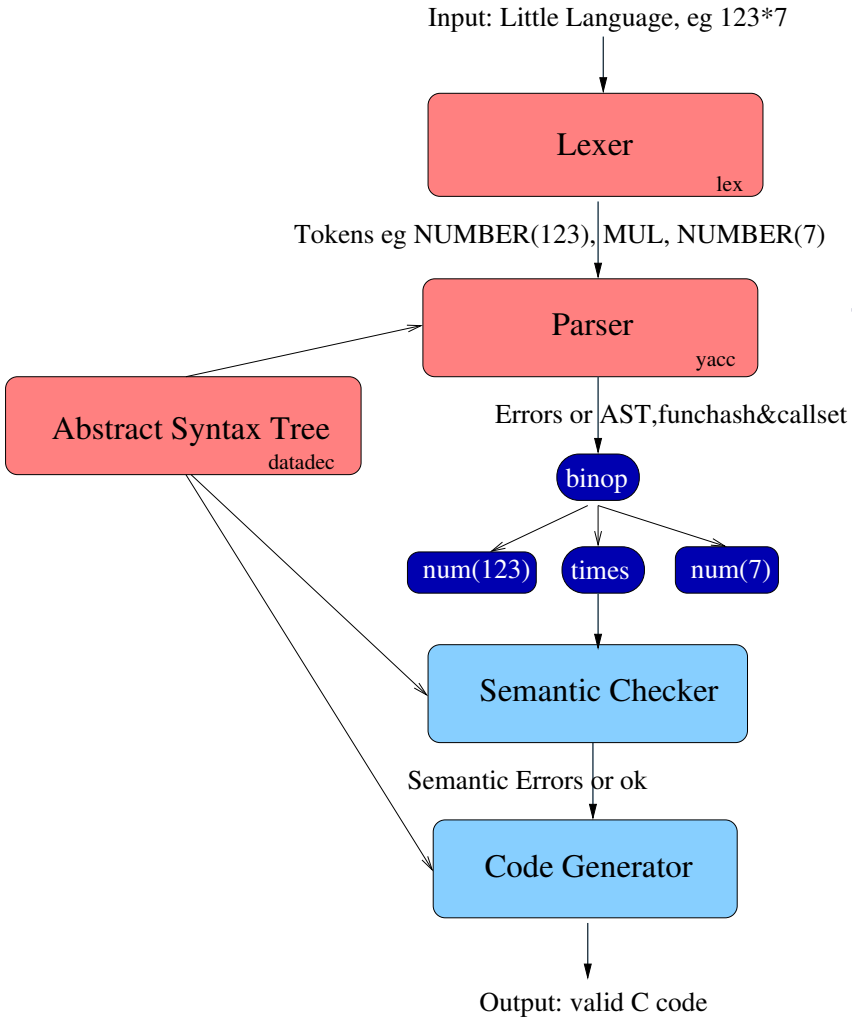
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.

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



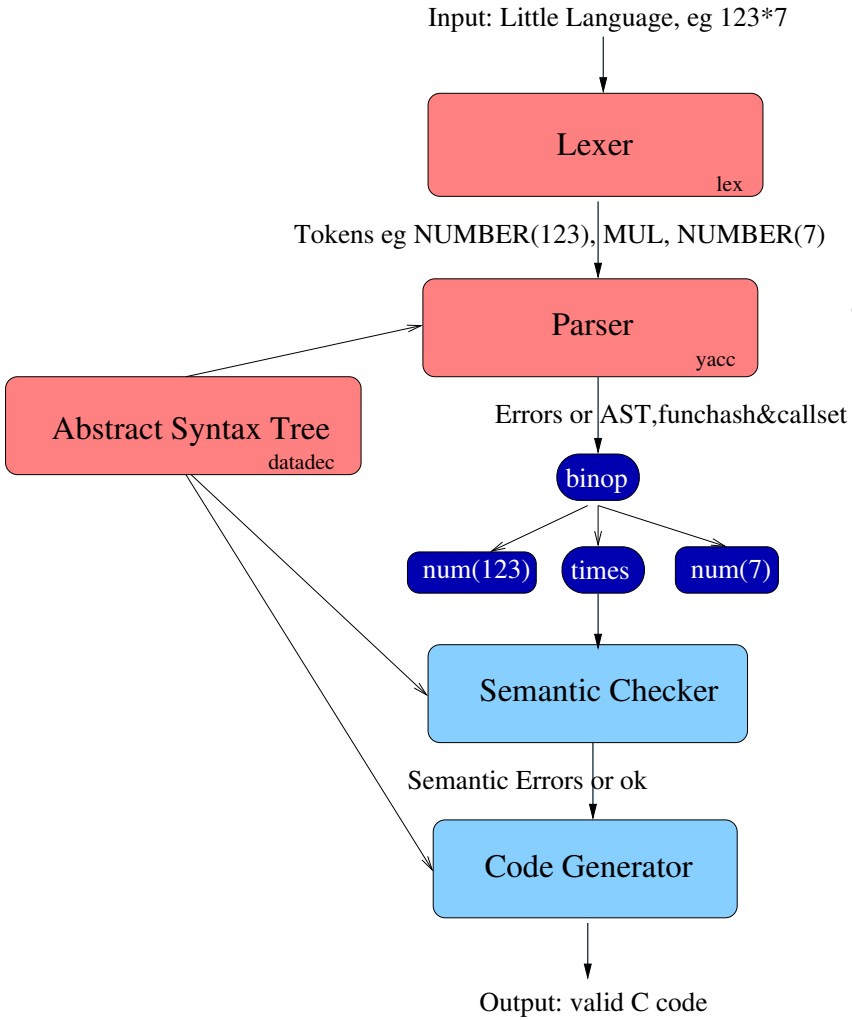
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar,

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



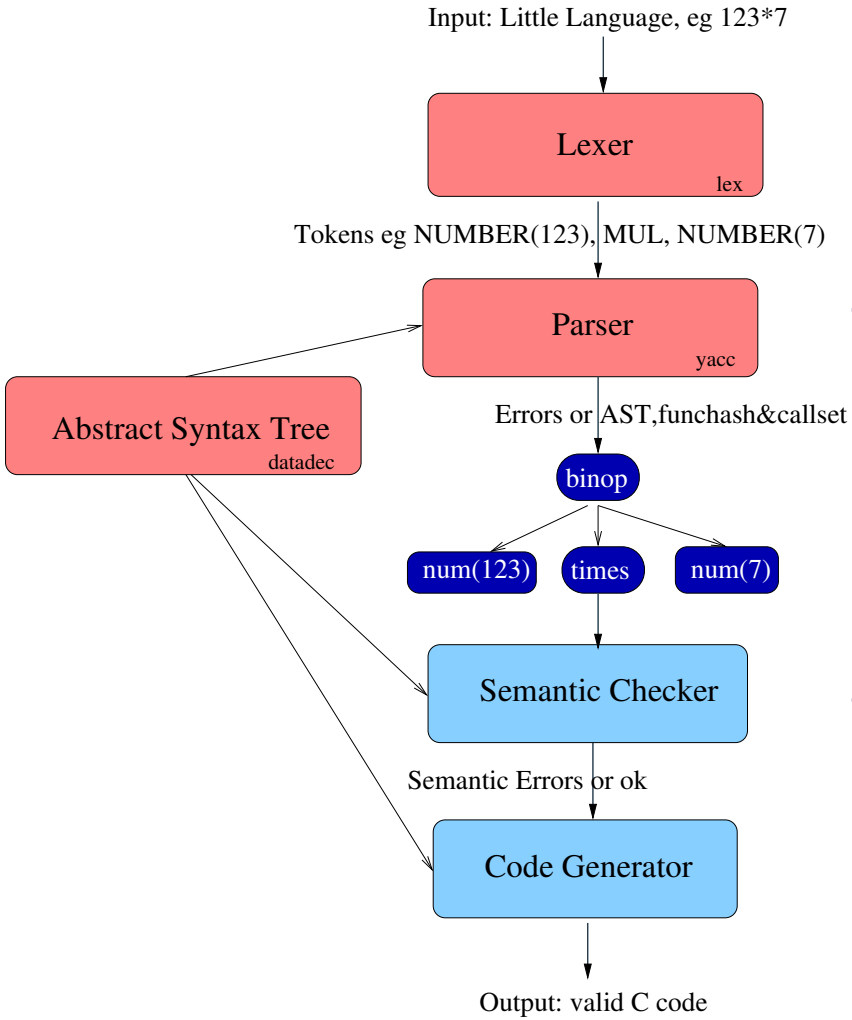
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar, builds an AST

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



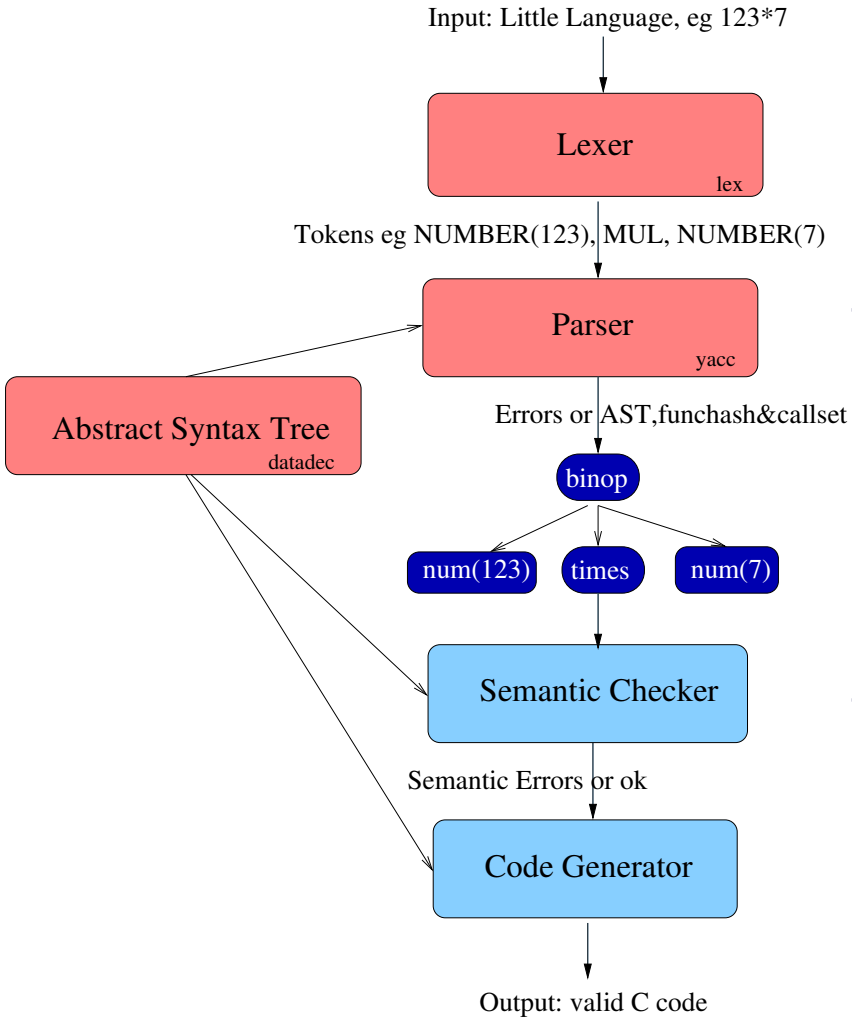
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar, builds an AST and builds funchash and callset (not shown).

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



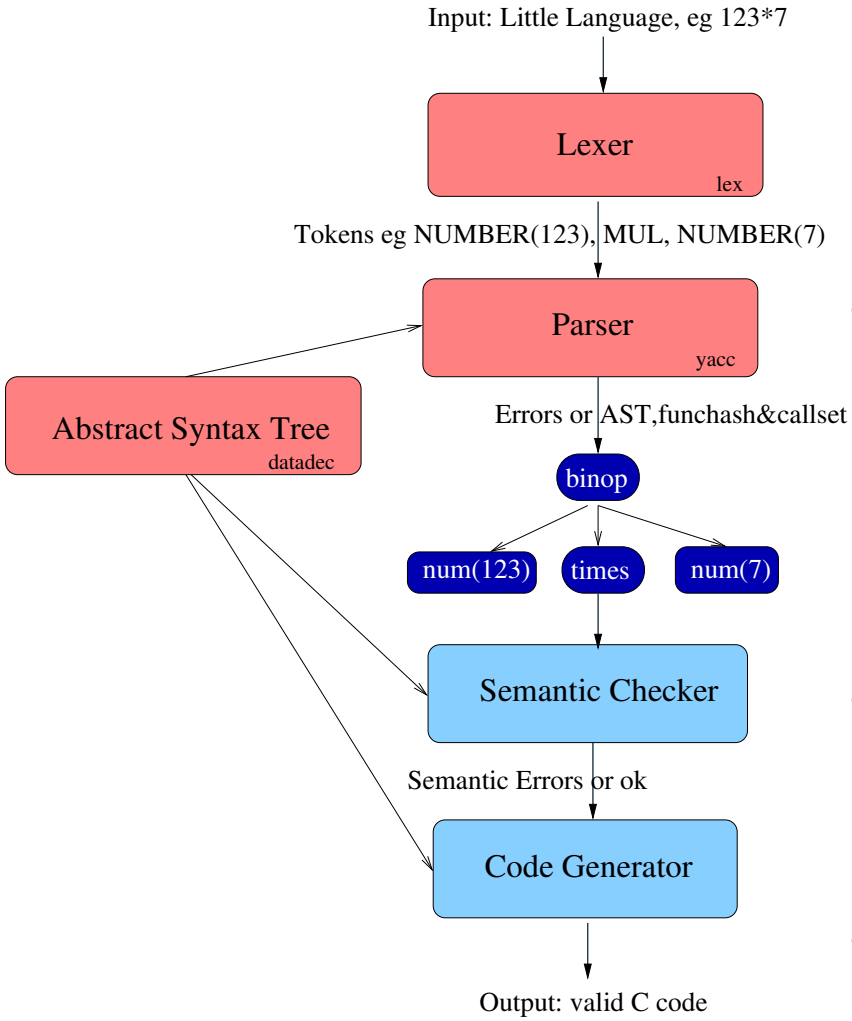
- Our Lexer (constructed for us by Lex) turns our input (eg "123*7", possibly with whitespace) into a stream of tokens.
- Our Parser (constructed for us by Yacc) checks whether the token stream matches the grammar, builds an AST and builds funchash and callset (not shown).
- Our Semantic checker uses the AST, funchash and callset

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.

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.

- 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 05.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 05.ths-codegen, we have only written about 850 lines of code ourselves.
- After **datadec, tmpl, proto, yacc, 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 05.ths-codegen, we have only written about 850 lines of code ourselves.
- After **datadec, tmpl, proto, yacc, 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 05.ths-codegen, we have only written about 850 lines of code ourselves.
- After **datadec, tmpl, proto, yacc, 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 05.ths-codegen, we have only written about 850 lines of code ourselves.
- After **datadec, tmpl, proto, yacc, 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 05.ths-codegen, we have only written about 850 lines of code ourselves.
- After **datadec, tmpl, proto, yacc, 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 05.ths-codegen, we have only written about 850 lines of code ourselves.
- After **datadec, tmpl, proto, yacc, 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 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 tarball examples.

- Recently, I've been playing with a different approach:

- Recently, I've been playing with a different approach: Suppose instead of defining a complete little language, we want to add a single well-defined feature to a large language like C.

- Recently, I've been playing with a different 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 the previous lecture. From time to time I've thought that some sort of pattern match would be lovely.

- Recently, I've been playing with a different 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 the previous lecture. From time to time I've thought that some sort of pattern match would be lovely. 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);
        }
}
```

- Recently, I've been playing with a different 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 the previous lecture. From time to time I've thought that some sort of pattern match would be lovely. 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;
}
```

- 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);
}
```

- 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.

- 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 and graft our new feature into it.

- 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 and graft our new feature into it.
- But that sounds like hard work! We'd have to work out what assembly code (or intermediate code such as Register Transfer Code) to emit for our new features.

- 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?

- 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.

- 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!

- 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 less... scary way to do this?

- 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 less... scary way to do this? Yes there is!

- 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 less... scary way to do this? 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.

- 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 less... scary way to do this? 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 06.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 06.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);
        }
}
```

- In 06.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.

- In 06.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:

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

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

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

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

- 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.

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

- 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!

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

- 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.

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

- 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.

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

- 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/`

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

- 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/`

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

- 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!

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

- 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!