C Programming Tools: Part 4

Building Lexers and Parsers

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

Dept of Computing,
Imperial College London

The handout and tarball are available on materials.doc.ic.ac.uk and at:
`http://www.doc.ic.ac.uk/~dcw/c-tools-2021/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.
- Most of those tools were Code Generators - Code that Writes Code (Tip 29).

- Last lecture, we started building our own tools when necessary, at a range of scales from tiny to large.
- Most of those tools were Code Generators - Code that Writes Code (Tip 29).
- 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.
- Most of those tools were Code Generators - Code that Writes Code (Tip 29).
- A Code Generator defines some Little Language and then translates that into some other form - eg valid C source code.
- The main topic of this lecture is to 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.

- The first part of writing any Code Generator is to build a lexical analyser (aka a Lexer) and a Parser for your little language.
- Everyone should write a couple of lexers and parsers by hand to get the hang of them, but..

- The first part of writing any Code Generator is to build a lexical analyser (aka a Lexer) and a Parser for your little language.
- Everyone should write a couple of lexers and parsers by hand to get the hang of them, but.. After that, it's just boring gruntwork - so use tools:

- The first part of writing any Code Generator is to build a lexical analyser (aka a Lexer) and a Parser for your little language.
- Everyone should write a couple of lexers and parsers by hand to get the hang of them, but.. After that, it's just boring gruntwork - so use tools:
- Lex generates C code (a Lexer) from declarative definitions of lexical tokens, and how to recognise them in the input.

- The first part of writing any Code Generator is to build a lexical analyser (aka a Lexer) and a Parser for your little language.
- Everyone should write a couple of lexers and parsers by hand to get the hang of them, but.. After that, it's just boring gruntwork - so use tools:
- Lex generates C code (a Lexer) from declarative definitions of lexical tokens, and how to recognise them in the input.
- 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.

- The first part of writing any Code Generator is to build a lexical analyser (aka a Lexer) and a Parser for your little language.
- Everyone should write a couple of lexers and parsers by hand to get the hang of them, but.. After that, it's just boring gruntwork - so use tools:
- Lex generates C code (a Lexer) from declarative definitions of lexical tokens, and how to recognise them in the input.
- 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.
- The actions you tell Yacc to take when constructs are parsed can do anything, but typically they build an Abstract Syntax Tree or AST, aka the parse tree. This is an internal form of the little language input, used by later phases of compilation (semantic checking and code generation) - implemented as AST tree walkers.

- The first part of writing any Code Generator is to build a lexical analyser (aka a Lexer) and a Parser for your little language.
- Everyone should write a couple of lexers and parsers by hand to get the hang of them, but.. After that, it's just boring gruntwork - so use tools:
- Lex generates C code (a Lexer) from declarative definitions of lexical tokens, and how to recognise them in the input.
- 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.
- The actions you tell Yacc to take when constructs are parsed can do anything, but typically they build an Abstract Syntax Tree or AST, aka the parse tree. This is an internal form of the little language input, used by later phases of compilation (semantic checking and code generation) - implemented as AST tree walkers.
- Note: Datadec is the perfect tool to generate ASTs.

- What Little Language shall we use as an example?

- What Little Language shall we use as an example?
- Let's start with integer constant expressions such as `3*(10+eek*(123/3) mod 7)`.

- What Little Language shall we use as an example?
- Let's start with integer constant expressions such as `3*(10+eek*(123/3) mod 7)`.
- Looking first at the lexical level, the basic 'tokens' needed are:
  - Numeric constants (eg '123').
  - Identifiers - named constants (eg 'eek') whose values are defined elsewhere.
  - Various one-character operators (eg. '(', '+', '*', ')' etc).
  - A Haskell-inspired keyword 'mod' (i.e. modulus, '%' in C terms).

- What Little Language shall we use as an example?
- Let's start with integer constant expressions such as `3*(10+eek*(123/3) mod 7)`.
- Looking first at the lexical level, the basic 'tokens' needed are:
  - Numeric constants (eg '123').
  - Identifiers - named constants (eg 'eek') whose values are defined elsewhere.
  - Various one-character operators (eg. '(', '+', '*', ')' etc).
  - A Haskell-inspired keyword 'mod' (i.e. modulus, '%' in C terms).
- Two of those tokens have associated values:
  - A numeric constant has an associated integer value - which particular number we have seen, eg. 3, 10, 123 etc.

- What Little Language shall we use as an example?
- Let's start with integer constant expressions such as `3*(10+eek*(123/3) mod 7)`.
- Looking first at the lexical level, the basic 'tokens' needed are:
  - Numeric constants (eg '123').
  - Identifiers - named constants (eg 'eek') whose values are defined elsewhere.
  - Various one-character operators (eg. '(', '+', '*', ')' etc).
  - A Haskell-inspired keyword 'mod' (i.e. modulus, '%' in C terms).
- Two of those tokens have associated values:
  - A numeric constant has an associated integer value - which particular number we have seen, eg. 3, 10, 123 etc.
  - An identifier has an associated string (char *) - the actual name of the identifier that we've seen, eg "eek".

- What Little Language shall we use as an example?
- Let's start with integer constant expressions such as `3*(10+eek*(123/3) mod 7)`.
- Looking first at the lexical level, the basic 'tokens' needed are:
  - Numeric constants (eg '123').
  - Identifiers - named constants (eg 'eek') whose values are defined elsewhere.
  - Various one-character operators (eg. '(', '+', '*', ')' etc).
  - A Haskell-inspired keyword 'mod' (i.e. modulus, '%' in C terms).
- Two of those tokens have associated values:
  - A numeric constant has an associated integer value - which particular number we have seen, eg. 3, 10, 123 etc.
  - An identifier has an associated string (char *) - the actual name of the identifier that we've seen, eg "eek".
- If we were writing the lexer in Haskell, then a token would be represented by the following inductive data type:

```
token =  PLUS or MINUS or MOD or MUL or DIV or OPEN or CLOSE
      or IDENT(string s) or NUMBER(int n) or TOKERR;
```

- To represent this in C, we define the tokens themselves as integer constants (using any distinct, non zero values we like):

```
#define PLUS    1
#define MINUS   2
#define MUL     3
#define DIV     4
#define MOD     5
#define OPEN    6
#define CLOSE   7
#define TOKERR  8
#define NUMBER  9
#define IDENT  10
```

- To represent this in C, we define the tokens themselves as integer constants (using any distinct, non zero values we like):

```
#define PLUS    1
#define MINUS   2
#define MUL     3
#define DIV     4
#define MOD     5
#define OPEN    6
#define CLOSE   7
#define TOKERR  8
#define NUMBER  9
#define IDENT  10
```

- To represent the associated values (an integer n and a char *s), in a Lex-compatible way, we define a union type and a variable of that type:

```
typedef union
{
      int n;
      char *s;
} YYSTYPE;

extern YYSTYPE yylval;     // "lexical value" associated with current token, if any
```

- To represent this in C, we define the tokens themselves as integer constants (using any distinct, non zero values we like):

```
#define PLUS     1
#define MINUS    2
#define MUL      3
#define DIV      4
#define MOD      5
#define OPEN     6
#define CLOSE    7
#define TOKERR   8
#define NUMBER   9
#define IDENT   10
```

- To represent the associated values (an integer n and a char *s), in a Lex-compatible way, we define a union type and a variable of that type:

```
typedef union
{
      int n;
      char *s;
} YYSTYPE;

extern YYSTYPE yylval;     // "lexical value" associated with current token, if any
```

- We put these definitions in lexsupport.h, along with a prototype for a function to print out the current token:

```
extern void print_token( FILE * out, int tok );
```

- In lexsupport.c we write:

```
#include <stdio.h>
#include "lexsupport.h"

YYSTYPE yylval;     // value associated with current token, if any

/*
      Print token <tok> and it's associated
      value from yylval (if any) to <out>
 */
void print_token( FILE *out, int tok )
{
      switch( tok )
      {
      case PLUS:      fputc( '+', out ); break;
      case MINUS:     fputc( '-', out ); break;
      case MUL:       fputc( '*', out ); break;
      case DIV:       fputc( '/', out ); break;
      case MOD:       fputs( "mod", out ); break;
      case OPEN:      fputc( '(', out ); break;
      case CLOSE:     fputc( ')', out ); break;
      case TOKERR:    fputs( "<UNKNOWN TOKEN>", out ); break;
      case NUMBER:    fprintf( out, "number(%d)", yylval.n ); break;
      case IDENT:     fprintf( out, "ident(%s)", yylval.s ); break;
      default:        fprintf( out, "<IMPOSSIBLE TOKEN %d>", tok );
      }
}
```

(and a small irritating Lex-support function called yywrap, not shown).

- Now, to define the lexical rules for our tokens, Lex allows us to specify regular expression/action pairs:

```
\+                      return PLUS;
-                       return MINUS;
\*                      return MUL;
\/                      return DIV;
\(                      return OPEN;
\)                      return CLOSE;
mod                     return MOD;
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext);return IDENT;
[ \t\n]+                /* ignore whitespace */;
.                       return TOKERR;
```

- Most Lex rules are obvious, essentially when you match this string, return this token value. Note that regular expression rules mean that special characters like `+`, `*`, `/`, `(` and `)` need to be back-slashed.

- Now, to define the lexical rules for our tokens, Lex allows us to specify regular expression/action pairs:

```
\+                      return PLUS;
-                       return MINUS;
\*                      return MUL;
\/                      return DIV;
\(                      return OPEN;
\)                      return CLOSE;
mod                     return MOD;
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext);return IDENT;
[ \t\n]+                /* ignore whitespace */;
.                       return TOKERR;
```

- Most Lex rules are obvious, essentially when you match this string, return this token value. Note that regular expression rules mean that special characters like `+`, `*`, `/`, `(` and `)` need to be back-slashed.
- A few rules are more complex - the rule `[ \t\n]+   /* ignore whitespace */;` matches any number of adjacent space, tab and newline characters, then executes the empty action, leaving us still in the Lexer trying to find the next token, looking for any pattern matches.

- Now, to define the lexical rules for our tokens, Lex allows us to specify regular expression/action pairs:

```
\+                  return PLUS;
-                   return MINUS;
\*                  return MUL;
\/                  return DIV;
\(                  return OPEN;
\)                  return CLOSE;
mod                 return MOD;
[0-9]+              yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*      yylval.s=strdup(yytext);return IDENT;
[ \t\n]+            /* ignore whitespace */;
.                   return TOKERR;
```

- Most Lex rules are obvious, essentially when you match this string, return this token value. Note that regular expression rules mean that special characters like `+`, `*`, `/`, `(` and `)` need to be back-slashed.
- A few rules are more complex - the rule `[ \t\n]+   /* ignore whitespace */;` matches any number of adjacent space, tab and newline characters, then executes the empty action, leaving us still in the Lexer trying to find the next token, looking for any pattern matches.
- Next, let's look at the NUMBER rule:

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

- Now, to define the lexical rules for our tokens, Lex allows us to specify regular expression/action pairs:

```
\+                      return PLUS;
-                       return MINUS;
\*                      return MUL;
\/                      return DIV;
\(                      return OPEN;
\)                      return CLOSE;
mod                     return MOD;
[0-9]+                  yylval.n=atoi(yytext); return NUMBER;
[a-z][a-z0-9]*          yylval.s=strdup(yytext);return IDENT;
[ \t\n]+                /* ignore whitespace */;
.                       return TOKERR;
```

- Most Lex rules are obvious, essentially when you match this string, return this token value. Note that regular expression rules mean that special characters like `+`, `*`, `/`, `(` and `)` need to be back-slashed.
- A few rules are more complex - the rule `[ \t\n]+    /* ignore whitespace */;` matches any number of adjacent space, tab and newline characters, then executes the empty action, leaving us still in the Lexer trying to find the next token, looking for any pattern matches.
- Next, let's look at the NUMBER rule:

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

The regex pattern `[0-9]+` represents an arbitrarily long sequence of one or more adjacent decimal digits.

- Looking at the action `yylval.n=atoi(yytext); return NUMBER`, we wonder what yytext is?

- Looking at the action `yylval.n=atoi(yytext); return NUMBER`, we wonder what yytext is?
- When a Lex pattern matches the first few characters in the unconsumed input, the lexer consumes the matching chunk of input, copying it into a string called yytext.

- Looking at the action `yylval.n=atoi(yytext); return NUMBER`, we wonder what yytext is?
- When a Lex pattern matches the first few characters in the unconsumed input, the lexer consumes the matching chunk of input, copying it into a string called yytext.
- So, when the regex [0-9]+ has matched, the longest digit sequence found in the input is stored in yytext.

- Looking at the action `yylval.n=atoi(yytext); return NUMBER`, we wonder what yytext is?
- When a Lex pattern matches the first few characters in the unconsumed input, the lexer consumes the matching chunk of input, copying it into a string called yytext.
- So, when the regex [0-9]+ has matched, the longest digit sequence found in the input is stored in yytext.
- Then our Lex action runs: it extracts the integer value via atoi(yytext), stores it in yylval.n (the integer associated with a NUMBER token) and returns NUMBER.

- Looking at the action `yylval.n=atoi(yytext); return NUMBER`, we wonder what yytext is?
- When a Lex pattern matches the first few characters in the unconsumed input, the lexer consumes the matching chunk of input, copying it into a string called yytext.
- So, when the regex [0−9]+ has matched, the longest digit sequence found in the input is stored in yytext.
- Then our Lex action runs: it extracts the integer value via atoi(yytext), stores it in yylval.n (the integer associated with a NUMBER token) and returns NUMBER.
- So, for example, if the lexer is called to deliver the next token and the next few characters of input are:

  ```
  12345*eek123 mod (x+77)
  ```

  the lexer sees that the input 12345 matches [0−9]+, so 12345 is consumed from the input, yytext is set to "12345", yylval.n is set to 12345, and the lexer returns NUMBER.

- Looking at the action `yylval.n=atoi(yytext); return NUMBER`, we wonder what yytext is?
- When a Lex pattern matches the first few characters in the unconsumed input, the lexer consumes the matching chunk of input, copying it into a string called yytext.
- So, when the regex `[0-9]+` has matched, the longest digit sequence found in the input is stored in yytext.
- Then our Lex action runs: it extracts the integer value via atoi(yytext), stores it in yylval.n (the integer associated with a NUMBER token) and returns NUMBER.
- So, for example, if the lexer is called to deliver the next token and the next few characters of input are:

  ```
  12345*eek123 mod (x+77)
  ```

  the lexer sees that the input 12345 matches `[0-9]+`, so 12345 is consumed from the input, yytext is set to "12345", yylval.n is set to 12345, and the lexer returns NUMBER.
- The unconsumed input is now: `*eek123 mod (x+77)`

- Similarly, when we look at the IDENT rule:

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

- Similarly, when we look at the IDENT rule:

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

- The regex represents a lower case letter (`[a-z]`) followed by zero or more lower case letters or digits (`[a-z0-9]*`), ie. a lower case alphanumeric string.

- Similarly, when we look at the IDENT rule:

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

- The regex represents a lower case letter (`[a-z]`) followed by zero or more lower case letters or digits (`[a-z0-9]*`), ie. a lower case alphanumeric string.
- When the pattern matches, the longest alphanumeric sequence found at the front of the input is stored in yytext - and consumed from the input.
- We strdup(yytext) to give ourselves a long-lived copy of the string, storing that in yylval.s, and then return IDENT.

- Similarly, when we look at the IDENT rule:

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

- The regex represents a lower case letter (`[a-z]`) followed by zero or more lower case letters or digits (`[a-z0-9]*`), ie. a lower case alphanumeric string.
- When the pattern matches, the longest alphanumeric sequence found at the front of the input is stored in yytext - and consumed from the input.
- We strdup(yytext) to give ourselves a long-lived copy of the string, storing that in yylval.s, and then return IDENT.
- For example, if the unconsumed input is: `eek123 mod (x+77)`

- Similarly, when we look at the IDENT rule:

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

- The regex represents a lower case letter (`[a-z]`) followed by zero or more lower case letters or digits (`[a-z0-9]*`), ie. a lower case alphanumeric string.
- When the pattern matches, the longest alphanumeric sequence found at the front of the input is stored in yytext - and consumed from the input.
- We strdup(yytext) to give ourselves a long-lived copy of the string, storing that in yylval.s, and then return IDENT.
- For example, if the unconsumed input is: `eek123 mod (x+77)`
- Then eek123 is consumed from the input, yytext is set to "eek123", which is then duplicated and stored in yylval.s, leaving `mod (x+77)` unconsumed.

- Similarly, when we look at the IDENT rule:

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

- The regex represents a lower case letter (`[a-z]`) followed by zero or more lower case letters or digits (`[a-z0-9]*`), ie. a lower case alphanumeric string.
- When the pattern matches, the longest alphanumeric sequence found at the front of the input is stored in yytext - and consumed from the input.
- We strdup(yytext) to give ourselves a long-lived copy of the string, storing that in yylval.s, and then return IDENT.
- For example, if the unconsumed input is: `eek123 mod (x+77)`
- Then eek123 is consumed from the input, yytext is set to "eek123", which is then duplicated and stored in yylval.s, leaving ` mod (x+77)` unconsumed.
- Note that Lex automatically handles overlapping patterns - the keyword mod is not confused with an identifier, despite the string mod also matching a lower case letter (m) followed by zero or more letters or digits (od).

- Similarly, when we look at the IDENT rule:

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

- The regex represents a lower case letter (`[a-z]`) followed by zero or more lower case letters or digits (`[a-z0-9]*`), ie. a lower case alphanumeric string.
- When the pattern matches, the longest alphanumeric sequence found at the front of the input is stored in yytext - and consumed from the input.
- We strdup(yytext) to give ourselves a long-lived copy of the string, storing that in yylval.s, and then return IDENT.
- For example, if the unconsumed input is: `eek123 mod (x+77)`
- Then eek123 is consumed from the input, yytext is set to "eek123", which is then duplicated and stored in yylval.s, leaving ` mod (x+77)` unconsumed.
- Note that Lex automatically handles overlapping patterns - the keyword mod is not confused with an identifier, despite the string mod also matching a lower case letter (m) followed by zero or more letters or digits (od).
- See lexer.l in 01.expr-lexonly 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.

- We complete the example with a main program mainprog.c, that repeatedly calls the yylex() function that Lex generates, and prints out each token that it finds:

```
int main( int argc, char **argv )
{
	int tok;
	while( (tok=yylex()) != 0 )
	{
		printf( "token: " );
		print_token( stdout, tok );
		putchar( '\n' );
	}
	yylex_destroy();
	return 0;
}
```

- We complete the example with a main program mainprog.c, that repeatedly calls the yylex() function that Lex generates, and prints out each token that it finds:

```
int main( int argc, char **argv )
{
      int tok;
      while( (tok=yylex()) != 0 )
      {
              printf( "token: " );
              print_token( stdout, tok );
              putchar( '\n' );
      }
      yylex_destroy();
      return 0;
}
```

- You'll find all these files in the 01.expr-lexonly directory, together with a Makefile to compile everything up. Type `make` and you're left with the executable `lextest`, which reads tokens from standard input.

- We complete the example with a main program mainprog.c, that repeatedly calls the yylex() function that Lex generates, and prints out each token that it finds:

```
int main( int argc, char **argv )
{
        int tok;
        while( (tok=yylex()) != 0 )
        {
                printf( "token: " );
                print_token( stdout, tok );
                putchar( '\n' );
        }
        yylex_destroy();
        return 0;
}
```

- You'll find all these files in the 01.expr-lexonly directory, together with a Makefile to compile everything up. Type `make` and you're left with the executable `lextest`, which reads tokens from standard input. Run it with input:

```
12345*eek123 mod (x+77)
```

and it generates output:

```
token: number(12345)          token: ident(x)
token: *                      token: +
token: ident(eek123)          token: number(77)
token: mod                    token: )
token: (
```

- Turning to the parser that Yacc is about to generate for us, this parser has two tasks:

- Turning to the parser that Yacc is about to generate for us, this parser has two tasks:
  - First, to check that this sequence of tokens generated by the lexer is valid under the grammatical rules we tell it.

- Turning to the parser that Yacc is about to generate for us, this parser has two tasks:
  - First, to check that this sequence of tokens generated by the lexer is valid under the grammatical rules we tell it.
  - Second, if it is valid, to generate an Abstract Syntax Tree representation of it.

- Turning to the parser that Yacc is about to generate for us, this parser has two tasks:
  - First, to check that this sequence of tokens generated by the lexer is valid under the grammatical rules we tell it.
  - Second, if it is valid, to generate an Abstract Syntax Tree representation of it.
- Our Abstract Syntax is most usefully defined as a series of Haskell-style inductive data types, specified (of course!) in a Datadec input file called types.in:

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

- Turning to the parser that Yacc is about to generate for us, this parser has two tasks:
  - First, to check that this sequence of tokens generated by the lexer is valid under the grammatical rules we tell it.
  - Second, if it is valid, to generate an Abstract Syntax Tree representation of it.
- Our Abstract Syntax is most usefully defined as a series of Haskell-style inductive data types, specified (of course!) in a Datadec input file called types.in:

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

- So, our parser's main job is to build an Abstract expr tree from our token stream.

- Turning to the parser that Yacc is about to generate for us, this parser has two tasks:
  - First, to check that this sequence of tokens generated by the lexer is valid under the grammatical rules we tell it.
  - Second, if it is valid, to generate an Abstract Syntax Tree representation of it.
- Our Abstract Syntax is most usefully defined as a series of Haskell-style inductive data types, specified (of course!) in a Datadec input file called types.in:

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

- So, our parser's main job is to build an Abstract expr tree from our token stream.
- To generate the parser, we provide a quite complicated Yacc input file called parser.y.

- parser.y starts with a long prelude of plain C code:

```
%{
// some includes and externs..

expr ast = NULL;
int yyerrors = 0;

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

- parser.y starts with a long prelude of plain C code:

```
%{
// some includes and externs..

expr ast = NULL;
int yyerrors = 0;

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

- The parser calls the yyerror() function to report parse errors. Note the use of the current source line number yylineno, which yylex() automatically keeps track of.

- parser.y starts with a long prelude of plain C code:

```
%{
// some includes and externs..

expr ast = NULL;
int yyerrors = 0;

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

- The parser calls the yyerror() function to report parse errors. Note the use of the current source line number yylineno, which yylex() automatically keeps track of.
- Also note that we count the total number of parse errors in yyerrors.

- parser.y starts with a long prelude of plain C code:

```
%{
// some includes and externs..

expr ast = NULL;
int yyerrors = 0;

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

- The parser calls the yyerror() function to report parse errors. Note the use of the current source line number yylineno, which yylex() automatically keeps track of.
- Also note that we count the total number of parse errors in yyerrors.
- The variable definition:

```
expr ast = NULL;
```

defines the variable ast that we will use to store the AST representation (an expr) of the whole integer expression after a successful parse.

- Next parser.y contains a `%union` declaration listing all possible types of data associated with tokens (and parse rules):

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

- Next parser.y contains a `%union` declaration listing all possible types of data associated with tokens (and parse rules):

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

- Yacc auto-generates the YYSTYPE union declaration and the yylval variable (that we previously defined in lexsupport.h) from this information, and places it in parser.h.

- Next parser.y contains a `%union` declaration listing all possible types of data associated with tokens (and parse rules):

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

- Yacc auto-generates the YYSTYPE union declaration and the yylval variable (that we previously defined in lexsupport.h) from this information, and places it in parser.h.
- The Lex prelude is modified slightly to include parser.h rather than lexsupport.h, allowing Lex rules to store values in yylval.n and yylval.s as before.
- Our `%union` also contains field `expr e`. We'll come back to that.

- Next parser.y contains a %union declaration listing all possible types of data associated with tokens (and parse rules):

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

- Yacc auto-generates the YYSTYPE union declaration and the yylval variable (that we previously defined in lexsupport.h) from this information, and places it in parser.h.
- The Lex prelude is modified slightly to include parser.h rather than lexsupport.h, allowing Lex rules to store values in yylval.n and yylval.s as before.
- Our %union also contains field expr e. We'll come back to that.
- Below the %union we see a list of all the tokens, first those without associated values:

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

- Next parser.y contains a `%union` declaration listing all possible types of data associated with tokens (and parse rules):

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

- Yacc auto-generates the YYSTYPE union declaration and the yylval variable (that we previously defined in lexsupport.h) from this information, and places it in parser.h.
- The Lex prelude is modified slightly to include parser.h rather than lexsupport.h, allowing Lex rules to store values in yylval.n and yylval.s as before.
- Our `%union` also contains field `expr e`. We'll come back to that.
- Below the `%union` we see a list of all the tokens, first those without associated values:

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

- Then we list those tokens with associated values:

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

- These tell Yacc that a NUMBER token has an associated int n value, and an IDENT token has an associated char *s value.
- So, the lexer deposits the actual number seen in yylval.n and Yacc looks in yylval.n to retrieve that value later on.

- These tell Yacc that a NUMBER token has an associated int n value, and an IDENT token has an associated char *s value.
- So, the lexer deposits the actual number seen in yylval.n and Yacc looks in yylval.n to retrieve that value later on.
- Next, we tell Yacc that several parse rules also have associated values - the expr e field in the union, allowing those parse rules to build abstract expressions:

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

- These tell Yacc that a NUMBER token has an associated int n value, and an IDENT token has an associated char *s value.
- So, the lexer deposits the actual number seen in yylval.n and Yacc looks in yylval.n to retrieve that value later on.
- Next, we tell Yacc that several parse rules also have associated values - the expr e field in the union, allowing those parse rules to build abstract expressions:

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

- Next, we tell Yacc which rule the parser that it generates must attempt to parse, i.e. which is the whole input must match this rule. Here we call that start rule top:

```
%start top
```

- These tell Yacc that a NUMBER token has an associated int n value, and an IDENT token has an associated char *s value.
- So, the lexer deposits the actual number seen in yylval.n and Yacc looks in yylval.n to retrieve that value later on.
- Next, we tell Yacc that several parse rules also have associated values - the expr e field in the union, allowing those parse rules to build abstract expressions:

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

- Next, we tell Yacc which rule the parser that it generates must attempt to parse, i.e. which is the whole input must match this rule. Here we call that start rule top:

```
%start top
```

- The rest of parser.y lists the grammatical parse rules that define integer expressions (in BNF), and the corresponding tree-building actions to take when a rule matches. The first rule is:

```
%%
top          : expr            { ast = $1; }
             ;
```

- These tell Yacc that a NUMBER token has an associated int n value, and an IDENT token has an associated char *s value.
- So, the lexer deposits the actual number seen in yylval.n and Yacc looks in yylval.n to retrieve that value later on.
- Next, we tell Yacc that several parse rules also have associated values - the expr e field in the union, allowing those parse rules to build abstract expressions:

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

- Next, we tell Yacc which rule the parser that it generates must attempt to parse, i.e. which is the whole input must match this rule. Here we call that start rule top:

```
%start top
```

- The rest of parser.y lists the grammatical parse rules that define integer expressions (in BNF), and the corresponding tree-building actions to take when a rule matches. The first rule is:

```
%%
top             : expr              { ast = $1; }
                ;
```

So our parser must match the entire input - with none left over - as an expression. We'll discuss the action in a moment.

- The parse rules continue:

```
expr          : expr PLUS term   { $$ = mkplus( $1, $3 ); }
              | expr MINUS term  { $$ = mkminus( $1, $3 ); }
              | term             { $$ = $1; }
              ;
term          : term MUL factor  { $$ = mktimes( $1, $3 ); }
              | term DIV factor  { $$ = mkdivide( $1, $3 ); }
              | term MOD factor  { $$ = mkmod( $1, $3 ); }
              | factor           { $$ = $1; }
              ;
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- The parse rules continue:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
            | expr MINUS term  { $$ = mkminus( $1, $3 ); }
            | term             { $$ = $1; }
            ;
term        : term MUL factor  { $$ = mktimes( $1, $3 ); }
            | term DIV factor  { $$ = mkdivide( $1, $3 ); }
            | term MOD factor  { $$ = mkmod( $1, $3 ); }
            | factor           { $$ = $1; }
            ;
factor      : NUMBER           { $$ = expr_num($1); }
            | IDENT            { $$ = expr_id($1); }
            | OPEN expr CLOSE  { $$ = $2; }
            ;
```

- Looking just at the rules (ignoring the actions for a moment):
    - an expression is a list of one or more terms linked by PLUS/MINUS tokens,

- The parse rules continue:

```
expr          : expr PLUS term   { $$ = mkplus( $1, $3 ); }
              | expr MINUS term  { $$ = mkminus( $1, $3 ); }
              | term             { $$ = $1; }
              ;
term          : term MUL factor  { $$ = mktimes( $1, $3 ); }
              | term DIV factor  { $$ = mkdivide( $1, $3 ); }
              | term MOD factor  { $$ = mkmod( $1, $3 ); }
              | factor           { $$ = $1; }
              ;
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- Looking just at the rules (ignoring the actions for a moment):
  - an expression is a list of one or more terms linked by PLUS/MINUS tokens,
  - a term is a list of one or more factors linked by MUL/DIV/MOD tokens

- The parse rules continue:

```
expr          : expr PLUS term   { $$ = mkplus( $1, $3 ); }
              | expr MINUS term  { $$ = mkminus( $1, $3 ); }
              | term             { $$ = $1; }
              ;
term          : term MUL factor  { $$ = mktimes( $1, $3 ); }
              | term DIV factor  { $$ = mkdivide( $1, $3 ); }
              | term MOD factor  { $$ = mkmod( $1, $3 ); }
              | factor           { $$ = $1; }
              ;
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- Looking just at the rules (ignoring the actions for a moment):
  - an expression is a list of one or more terms linked by PLUS/MINUS tokens,
  - a term is a list of one or more factors linked by MUL/DIV/MOD tokens
  - and a factor is a numeric constant, an identifier or a bracketed sub-expression.

- The parse rules continue:

```
expr          : expr PLUS term   { $$ = mkplus( $1, $3 ); }
              | expr MINUS term  { $$ = mkminus( $1, $3 ); }
              | term             { $$ = $1; }
              ;
term          : term MUL factor  { $$ = mktimes( $1, $3 ); }
              | term DIV factor  { $$ = mkdivide( $1, $3 ); }
              | term MOD factor  { $$ = mkmod( $1, $3 ); }
              | factor           { $$ = $1; }
              ;
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- Looking just at the rules (ignoring the actions for a moment):
  - an expression is a list of one or more terms linked by PLUS/MINUS tokens,
  - a term is a list of one or more factors linked by MUL/DIV/MOD tokens
  - and a factor is a numeric constant, an identifier or a bracketed sub-expression.
- But what about the actions?

- Picking one of our parse rules/action pairs out, we see:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
```

- Picking one of our parse rules/action pairs out, we see:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
```

- This rule says that one syntactic form of an integer expression comprises a sub-expression, followed by a PLUS token ('+'), followed by a term.

- Picking one of our parse rules/action pairs out, we see:

```
expr          : expr PLUS term    { $$ = mkplus( $1, $3 ); }
```

- This rule says that one syntactic form of an integer expression comprises a sub-expression, followed by a PLUS token ('+'), followed by a term.
- Note that recursive rules in Yacc like this one must be written with the recursive invocation of expr first. Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you write the more intuitive: `expr : term PLUS expr`.

- Picking one of our parse rules/action pairs out, we see:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
```

- This rule says that one syntactic form of an integer expression comprises a sub-expression, followed by a PLUS token ('+'), followed by a term.
- Note that recursive rules in Yacc like this one must be written with the recursive invocation of expr first. Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you write the more intuitive: `expr : term PLUS expr`.
- When our `expr PLUS term` rule matches, the action is executed, with:
  - $1 set to the value (if any) associated with the sub-expr rule,
  - $2 set to the value (if any) associated with the PLUS token,
  - $3 set to the value (if any) associated with the term rule.

- Picking one of our parse rules/action pairs out, we see:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
```

- This rule says that one syntactic form of an integer expression comprises a sub-expression, followed by a PLUS token ('+'), followed by a term.
- Note that recursive rules in Yacc like this one must be written with the recursive invocation of expr first. Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you write the more intuitive: `expr : term PLUS expr`.
- When our `expr PLUS term` rule matches, the action is executed, with:
  - $1 set to the value (if any) associated with the sub-expr rule,
  - $2 set to the value (if any) associated with the PLUS token,
  - $3 set to the value (if any) associated with the term rule.
- Of course, we know that only expr and term have associated values, PLUS does not; so using $2 would be an error. We use $1 and $3 to call `mkplus( $1, $3 )`. mkplus() is:

```
expr mkplus( expr a, expr b ) { return expr_binop(a, arithop_plus(), b); }
```

- Picking one of our parse rules/action pairs out, we see:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
```

- This rule says that one syntactic form of an integer expression comprises a sub-expression, followed by a PLUS token ('+'), followed by a term.
- Note that recursive rules in Yacc like this one must be written with the recursive invocation of expr first. Yacc's algorithm can't handle it the other way round - Yacc will generate a fatal error if you write the more intuitive: `expr : term PLUS expr`.
- When our `expr PLUS term` rule matches, the action is executed, with:
  - $1 set to the value (if any) associated with the sub-expr rule,
  - $2 set to the value (if any) associated with the PLUS token,
  - $3 set to the value (if any) associated with the term rule.
- Of course, we know that only expr and term have associated values, PLUS does not; so using $2 would be an error. We use $1 and $3 to call `mkplus( $1, $3 )`. mkplus() is:

```
expr mkplus( expr a, expr b ) { return expr_binop(a, arithop_plus(), b); }
```

- Assigning that new expression to $$ sets the value associated with the whole expr rule, think of this as the rule return value.

- Let's look at the rest of the expr rules and actions:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
            | expr MINUS term  { $$ = mkminus( $1, $3 ); }
            | term             { $$ = $1; }
            ;
```

- We've explained the first one. The second is very similar: another syntactic form of expression comprises an expression followed by a MINUS token followed by a term. When that matches, $1 is the sub-expression's associated value and $3 the term's associated value. These are combined by `mkminus($1,$3)` and assigned to $$.

- Let's look at the rest of the expr rules and actions:

```
expr          : expr PLUS term   { $$ = mkplus( $1, $3 ); }
              | expr MINUS term  { $$ = mkminus( $1, $3 ); }
              | term             { $$ = $1; }
              ;
```

- We've explained the first one. The second is very similar: another syntactic form of expression comprises an expression followed by a MINUS token followed by a term. When that matches, \$1 is the sub-expression's associated value and \$3 the term's associated value. These are combined by `mkminus($1,$3)` and assigned to \$\$.
- mkminus() is: `expr mkminus( expr a, expr b ) { return expr_binop(a, arithop_minus(), b); }`

- Let's look at the rest of the expr rules and actions:

```
expr        : expr PLUS term   { $$ = mkplus( $1, $3 ); }
            | expr MINUS term  { $$ = mkminus( $1, $3 ); }
            | term             { $$ = $1; }
            ;
```

- We've explained the first one. The second is very similar: another syntactic form of expression comprises an expression followed by a MINUS token followed by a term. When that matches, \$1 is the sub-expression's associated value and \$3 the term's associated value. These are combined by `mkminus($1,$3)` and assigned to \$\$.
- mkminus() is: `expr mkminus( expr a, expr b ) { return expr_binop(a, arithop_minus(), b); }`
- The third rule is simpler: another form of an expression is a single term (with no additive operators such as PLUS or MINUS). In this case, we simply copy the term's associated value \$1 into \$\$.

- Let's look at the rest of the expr rules and actions:

```
expr          : expr PLUS term   { $$ = mkplus( $1, $3 ); }
              | expr MINUS term  { $$ = mkminus( $1, $3 ); }
              | term             { $$ = $1; }
              ;
```

- We've explained the first one. The second is very similar: another syntactic form of expression comprises an expression followed by a MINUS token followed by a term. When that matches, $1 is the sub-expression's associated value and $3 the term's associated value. These are combined by `mkminus($1,$3)` and assigned to $$.
- mkminus() is: `expr mkminus( expr a, expr b ) { return expr_binop(a, arithop_minus(), b); }`
- The third rule is simpler: another form of an expression is a single term (with no additive operators such as PLUS or MINUS). In this case, we simply copy the term's associated value $1 into $$.
- Terms are incredibly similar - but with the higher priority multiplicative operators, so we'll not bother to explain them.

- Factors are more interesting, and deserve an explanation:

```
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- So: a factor may be a plain integer constant (the NUMBER token), in which case we construct an `expr_num()` from the number's associated value `$1` - `yylval.n`.

- Factors are more interesting, and deserve an explanation:

```
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- So: a factor may be a plain integer constant (the NUMBER token), in which case we construct an `expr_num()` from the number's associated value $1 - `yylval.n`.
- Or a factor may be an identifier (the IDENT token), in which case we construct an `expr_id()` from the identifier's associated value $1 - `yylval.s`, a malloc()d string allocated by strdup().

- Factors are more interesting, and deserve an explanation:

```
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- So: a factor may be a plain integer constant (the NUMBER token), in which case we construct an `expr_num()` from the number's associated value $1 - `yylval.n`.
- Or a factor may be an identifier (the IDENT token), in which case we construct an `expr_id()` from the identifier's associated value $1 - `yylval.s`, a malloc()d string allocated by strdup().
- Finally, a factor may be a bracketed sub-expression, in which case we copy the associated value $2 of the sub-expression.

- Factors are more interesting, and deserve an explanation:

```
factor       : NUMBER             { $$ = expr_num($1); }
             | IDENT              { $$ = expr_id($1); }
             | OPEN expr CLOSE    { $$ = $2; }
             ;
```

- So: a factor may be a plain integer constant (the NUMBER token), in which case we construct an `expr_num()` from the number's associated value $1 - `yylval.n`.
- Or a factor may be an identifier (the IDENT token), in which case we construct an `expr_id()` from the identifier's associated value $1 - `yylval.s`, a malloc()d string allocated by strdup().
- Finally, a factor may be a bracketed sub-expression, in which case we copy the associated value $2 of the sub-expression.
- The top level (start) rule, top, has a subtly different action:

```
top          : expr               { ast = $1; };
```

- Factors are more interesting, and deserve an explanation:

```
factor        : NUMBER           { $$ = expr_num($1); }
              | IDENT            { $$ = expr_id($1); }
              | OPEN expr CLOSE  { $$ = $2; }
              ;
```

- So: a factor may be a plain integer constant (the NUMBER token), in which case we construct an `expr_num()` from the number's associated value $1 - `yylval.n`.
- Or a factor may be an identifier (the IDENT token), in which case we construct an `expr_id()` from the identifier's associated value $1 - `yylval.s`, a malloc()d string allocated by strdup().
- Finally, a factor may be a bracketed sub-expression, in which case we copy the associated value $2 of the sub-expression.
- The top level (start) rule, top, has a subtly different action:

```
top          : expr             { ast = $1; };
```

- When this matches the entire input, with no junk left following a valid expr, that final abstract expr is copied from $1 to the `expr ast` variable. This enables the final fully built expr AST to be extracted from Yacc's clutches and returned to us.

- Turn parser.y into a C module (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.

- Turn parser.y into a C module (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.
- One remaining point is where the AST builder functions like mkplus() are actually stored, we've shown a couple of individual ones, but not where they're stored. Each is a thin wrapper on top of the datadec-generated types:

```
expr mkplus( expr a, expr b )
{
	return expr_binop(a, arithop_plus(), b);
}
```

- Turn parser.y into a C module (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.
- One remaining point is where the AST builder functions like mkplus() are actually stored, we've shown a couple of individual ones, but not where they're stored. Each is a thin wrapper on top of the datadec-generated types:

```
expr mkplus( expr a, expr b )
{
	return expr_binop(a, arithop_plus(), b);
}
```

- This function and all it's friends (mkminus() etc) are very repetitive. In the previous lecture we wrote a tiny tmpl tool to generate such output, so let's reuse it! We generate these functions from the input file binfuncs.in:

```
TEMPLATE,expr mk<0>( expr a, expr b )\n{\n\treturn expr_binop(a, arithop_<0>(), b);\n}\n
plus
minus
times
divide
mod
```

- Turn parser.y into a C module (parser.c and parser.h) via: yacc -vd -o parser.c parser.y.
- One remaining point is where the AST builder functions like mkplus() are actually stored, we've shown a couple of individual ones, but not where they're stored. Each is a thin wrapper on top of the datadec-generated types:

```
expr mkplus( expr a, expr b )
{
	return expr_binop(a, arithop_plus(), b);
}
```

- This function and all it's friends (mkminus() etc) are very repetitive. In the previous lecture we wrote a tiny tmpl tool to generate such output, so let's reuse it! We generate these functions from the input file binfuncs.in:

```
TEMPLATE,expr mk<0>( expr a, expr b )\n{\n\treturn expr_binop(a, arithop_<0>(), b);\n}\n
plus
minus
times
divide
mod
```

- A tiny helper shell script mkmodule is run with binfuncs as it's argument, it uses tmpl to turn binfuncs.in into binfuncs.c, then uses my other tool proto to generate binfuncs.h from binfuncs.c.

- The 02.expressions directory also provides:
  - A main program (mainprog.c) which initialises the lexer, calls the parser, and prints out the AST (if parsing is successful) or prints error messages if not,

- The 02.expressions directory also provides:
  - A main program (mainprog.c) which initialises the lexer, calls the parser, and prints out the AST (if parsing is successful) or prints error messages if not,
  - A module called consthash that gives expressions the ability to use predefined named constants (such as powers of two), and command line arguments (arg1..argn), these constants are stored in a longhash (from the previous lecture's libADTs library),

- The 02.expressions directory also provides:
  - A main program (mainprog.c) which initialises the lexer, calls the parser, and prints out the AST (if parsing is successful) or prints error messages if not,
  - A module called consthash that gives expressions the ability to use predefined named constants (such as powers of two), and command line arguments (arg1..argn), these constants are stored in a longhash (from the previous lecture's libADTs library),
  - A module called eval that evaluates expressions, by walking expr ast, using consthash to look up identifiers, and

- The 02.expressions directory also provides:
  - A main program (mainprog.c) which initialises the lexer, calls the parser, and prints out the AST (if parsing is successful) or prints error messages if not,
  - A module called consthash that gives expressions the ability to use predefined named constants (such as powers of two), and command line arguments (arg1..argn), these constants are stored in a longhash (from the previous lecture's libADTs library),
  - A module called eval that evaluates expressions, by walking expr ast, using consthash to look up identifiers, and
  - A Makefile to build everything, using **lex, yacc, datadec** and **mkmodule** to generate the lexer, parser, types module and binfuncs module, and then compiles and links everything.

- The 02.expressions directory also provides:
  - A main program (mainprog.c) which initialises the lexer, calls the parser, and prints out the AST (if parsing is successful) or prints error messages if not,
  - A module called consthash that gives expressions the ability to use predefined named constants (such as powers of two), and command line arguments (arg1..argn), these constants are stored in a longhash (from the previous lecture's libADTs library),
  - A module called eval that evaluates expressions, by walking expr ast, using consthash to look up identifiers, and
  - A Makefile to build everything, using **lex, yacc, datadec** and **mkmodule** to generate the lexer, parser, types module and binfuncs module, and then compiles and links everything.
- Build by typing make. We end up with an expression parser, treebuilder and evaluator called expr, in which we only write about 350 lines of code. Give it a try!

- The 02.expressions directory also provides:
  - A main program (mainprog.c) which initialises the lexer, calls the parser, and prints out the AST (if parsing is successful) or prints error messages if not,
  - A module called consthash that gives expressions the ability to use predefined named constants (such as powers of two), and command line arguments (arg1..argn), these constants are stored in a longhash (from the previous lecture's libADTs library),
  - A module called eval that evaluates expressions, by walking expr ast, using consthash to look up identifiers, and
  - A Makefile to build everything, using **lex, yacc, datadec** and **mkmodule** to generate the lexer, parser, types module and binfuncs module, and then compiles and links everything.
- Build by typing make. We end up with an expression parser, treebuilder and evaluator called expr, in which we only write about 350 lines of code. Give it a try!
- So far, we've used all this heavy-duty technology to essentially build a 5 dollar calculator. Are you impressed?

- The 02.expressions directory also provides:
  - A main program (mainprog.c) which initialises the lexer, calls the parser, and prints out the AST (if parsing is successful) or prints error messages if not,
  - A module called consthash that gives expressions the ability to use predefined named constants (such as powers of two), and command line arguments (arg1..argn), these constants are stored in a longhash (from the previous lecture's libADTs library),
  - A module called eval that evaluates expressions, by walking expr ast, using consthash to look up identifiers, and
  - A Makefile to build everything, using **lex, yacc, datadec** and **mkmodule** to generate the lexer, parser, types module and binfuncs module, and then compiles and links everything.
- Build by typing make. We end up with an expression parser, treebuilder and evaluator called expr, in which we only write about 350 lines of code. Give it a try!
- So far, we've used all this heavy-duty technology to essentially build a 5 dollar calculator. Are you impressed? Weellll.. Perhaps not:-). But in the final lecture we'll see how to scale our input language up significantly.