CSC 448: Compilers

Lecture 7

Joseph Phillips
De Paul University

2015 May 12

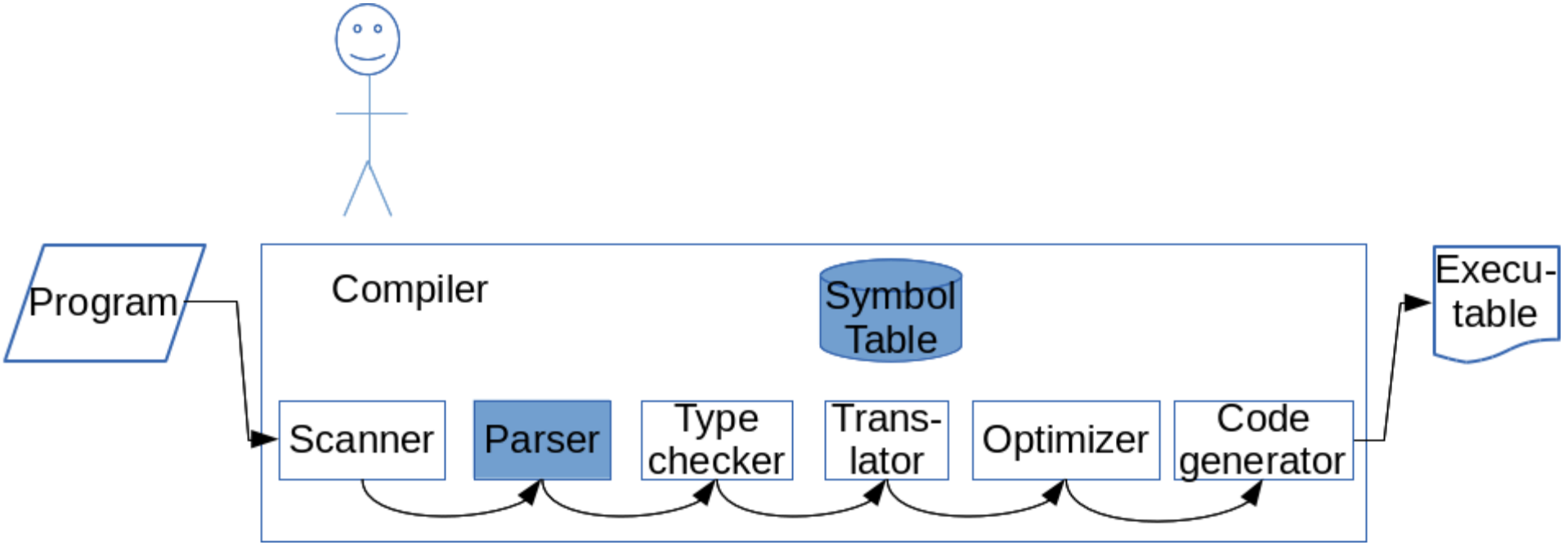
Reading

- Charles Fischer, Ron Cytron, Richard LeBlanc Jr. “Crafting a Compiler” Addison-Wesley. 2010.
  - Chapter 6: Bottom-Up Parsing
- Doug Brown, John R. Levine, Tony Mason. “lex & yacc, 2nd Ed.” O’Reilly Media. 1992
- John R. Levine. “flex & bison” O’Reilly Media. 2009.

Topics:

- Bottom-Up Parsing
- YACC/Bison

Overview:


Program
Compiler
Symbol Table
Scanner
Parser
Type checker
Trans-lator
Optimizer
Code generator
Execu-table

Remember our tables?

	$	#i	+	*	S	E	T
0	--	s st1	--	--	accept	s st2	s st3
1	r p5	--	r p5	r p5	--	--	--
2	s st4	--	s st5	--	--	--	--
3	r p3	--	r p3	s st6	--	--	--
4	r p1	--	--	--	--	--	--
5	--	s st1	--	--	--	--	s st7
6	--	s st8	--	--	--	--	--
7	r p2	--	r p2	s st6	--	--	--
8	r p4	--	r p4	r p4	--	--	--

They can be made by an algorithm, but it's nasty to implement

```
procedure computeLookahead()
  call buildItem PropGraph()
  call evalItemPropGraph()
end
procedure buildItemPropGraph()
  foreach s in States do
    foreach item in states do
      v := graph.addVert(s,item)
      itemFollow(v) := { }
  foreach p in prodFor(Start)
    itemFollow((StartState,S->·RHS(p)) := {$}
  foreach s in States
    foreach A ->α·Bγ in s do
      v := graph.findVert(s,A->α·Bγ)
      call graph.addEdge(v,(table[s][B],A->αB·γ))
etc.
```

Q: Hasn’t someone done that coding for me already?

A: Yes indeed! It’s called ***Bison***!

History of Bison

- YACC
  - "***Y***et ***A***nother ***C***ompiler ***C***ompiler"
  - By Stephen C. Johnson at ATT, early 1970s
  - Input:
    - Grammar (with embedded C code)
  - Output:
    - LALR(1) parser written in C that implements grammar
  - Plays well with *`lex`*
- GNU Bison
  - *Yak (yacc) vs. Bison, get it?*
  - By Robert Corbett and Richard Stallman, 1988-1990
  - Plays well with *`flex`*
  - Extensions include:
    - The ability to define sting constants as tokens

Our first YACC/Bison Program

- One program, three source files:
  - 1a.h: A header file with common inclusion and declarations
  - 1a.y: The YACC/Bison file with the grammar rules
  - 1a.lex: The Lex/Flex file that tokenizes

1a.h:

```
// 1a.h
#include        <stdlib.h>
#include        <stdio.h>
#include        <string.h>

#define LINE_LEN  256
#define YYSTYPE   double
extern  double    result;
extern  char*     textPtr;
extern  char*     textEndPtr;

extern  int       yyerror         (char *s);
extern  int       yylex           ();
```

1a.y (1)

```
%{
// 1a.y
// $ bison --verbose -d --debug 1a.y
// $ gcc 1a.tab.c -c
#include  "1a.h"
%}

%start      expr
%nonassoc   '+' '-' '*' '/' '(' ')' NUMBER
%nonassoc   ERROR

%%
```

1a.y (2)

```
expr    : expr '+' term
        {
          result = $$ = $1 + $3;
        }
      | expr '-' term
        {
          result = $$ = $1 - $3;
        }
      | term
        {
          result = $$ = $1;
        };
```

```
term    : term '*' factor
        {
          $$ = $1 * $3;
        }
      | term '/' factor
        {
          $$ = $1 / $3;
        }
      | factor
        {
          $$ = $1;
        };
```

1a.y (3)

```
factor  : '(' expr ')'
          {
            $$ = $2;
          }
        | NUMBER
          {
            $$ = $1;
          };

%%
double  result          = 0.0;
char*   textPtr         = NULL;
char*   textEndPtr      = NULL;

int     yyerror (char *cPtr)
{
  printf("%s, sorry!\n",cPtr);
  return(0);
}
```

```
int     main    (int argc, char* argv[])
{
  char  line[LINE_LEN];

  if  (argc >= 2)
    textPtr     = argv[1];
  else
  {
    printf("Please enter an expression: ");
    textPtr = fgets(line,LINE_LEN,stdin);
  }

  textEndPtr    = textPtr + strlen(textPtr);
  yyparse();
  printf("%g\n",result);
  return(EXIT_SUCCESS);
}
```

1a.lex (1)

```
%{
//  1a.lex
//  unix> flex -o 1a.c 1a.lex
//  unix> gcc 1a.c -c
//  unix> gcc -o 1a 1a.tab.o 1a.o

#include       "1a.h"
#include       "1a.tab.h"

#undef         YY_INPUT
#define        YY_INPUT(buffer,result,maxSize)       \
            { result = ourInput(buffer,maxSize); }

extern
int            ourInput(char* buffer, int maxSize);

#define        MIN(x,y)       (((x)<(y)) ? (x) : (y))

%}
```

```
%%

[ \t\n]     { /* ignore spaces */ }
[0-9]+|([0-9]*\.[0-9]+) {
       yylval = strtod(yytext,NULL);
       return(NUMBER);
                              }
\+   { return('+'); }
\-   { return('-'); }
\*   { return('*'); }
\/   { return('/'); }
\(   { return('('); }
\)   { return(')'); }
.    {

printf("What's'%c'?\n",yytext[0]);
       return(ERROR);
     }
```

1a.lex (2)

```
%%

int             ourInput(char* buffer, int maxSize)
{
  int   n       = MIN(maxSize,textEndPtr - textPtr);

  if  (n > 0)
  {
    memcpy(buffer,textPtr,n);
    textPtr     += n;
  }

  return(n);
}

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

Whew! That was work! How do we make it?

```
$ bison --verbose -d --debug la.y
  # Makes la.output la.tab.c la.tab.h
$ flex -o la.c la.lex
  # Makes la.c
$ gcc -o la la.tab.c la.c
  # Makes executable la
```

Now, run it!

It knows operator precedence: 4 + 8 * 2 = 4 + (8 * 2)

```
$ ./1a
Please enter an expression: 4 + 8 * 2
20
$ ./1a
Please enter an expression: (4 + 8) * 2
24
$ ./1a
```

It knows left-association: 6 - 3 - 2 = (6 - 3) - 2

```
Please enter an expression: 6 - 3 - 2
1
```

Okay, so it works, but how?

- First, let's remind ourselves how 1a.lex works:

```
%{
// C declarations
%}
%%
// Lexeme rules
%%
// C source code
```

1a.lex: C declarations

```
%{
//  1a.lex
//  unix> flex -o 1a.c 1a.lex
//  unix> gcc 1a.c -c
//  unix> gcc -o 1a 1a.tab.o 1a.o

#include  "1a.h"          // We wrote this
#include  "1a.tab.h"      // Where did this come from?  Stay tuned!

#undef    YY_INPUT        // Let's define our own input source
#define   YY_INPUT(buf,result,size)  { result = ourInput(buf,size); }
extern    int  ourInput(char* buffer, int maxSize);
#define   MIN(x,y)  (((x)<(y)) ? (x) : (y))  // A useful macro

%}
```

1a.lex: Lexeme rules

```
%%

[ \t\n]                    { /* ignore spaces */ }
[0-9]+|([0-9]*\.[0-9]+) {
                             yylval = strtod(yytext,NULL);
                                                  // HEY!  What's yylval?
                             return(NUMBER); // HEY!  What's NUMBER?
                           }
\+                         { return('+'); }  // Treat +-*/() as themselves
\-                         { return('-'); }
\*                         { return('*'); }
\/                         { return('/'); }
\(                         { return('('); }
\)                         { return(')'); }
.                          {
                             printf("What's'%c'?\n",yytext[0]);
                             return(ERROR);  // HEY!  What's ERROR?
                           }

%%
```

1a.lex: C code

```
// We'll get our input from textPtr,
// textEndPtr will tell us how much there is.
int     ourInput(char* buffer, int maxSize)
{
  int  n  = MIN(maxSize,textEndPtr - textPtr);
  if  (n > 0)
  {
    memcpy(buffer,textPtr,n);
    textPtr     += n;
  }
  return(n);
}

// Tells lex "There is no more input
//  after ourInput()"
int   yywrap ()   { return(1); }
```

Now on to YACC/Bison:

```
%{
// C declarations
%}
// Start symbol & token declarations
%%
// Grammar rules
%%
// C source code
```

1a.y: C declarations

```
%{
// $ bison --verbose -d --debug 1a.y
// $ gcc 1a.tab.c -c
#include  "1a.h"
%}

// (Kind of boring)
```

1a.y: Start non-terminal and terminal declarations

```
// Declare 'expr' will be our starting non-terminal
%start        expr

//  Declares several tokens . . .
//  including our mystery NUMBER and ERROR
%nonassoc     '+' '-' '*' '/' '(' ')' NUMBER
%nonassoc     ERROR
```

1a.y: Grammar rules

```
expr    : expr '+' term
          {
            result = $$ = $1 + $3;
          }
        | expr '-' term
          {
            result = $$ = $1 - $3;
          }
        | term
          {
            result = $$ = $1;
          };

// You've seen left-associative productions: E -> E '+' T | E '-' T | T
// $1 means "the value associated with the 1st RHS symbol"
// $2 means "the value associated with the 2nd RHS symbol"
// $$ means "the value associated with LHS symbol"
// Note 1: Bottom-up parsing.  Generally $$ = someFunction($1,..,$n)
// Note 2: Note syntax:  lhs : rhs11..rhs1M { } | rhs21..rhs2N { } ;
```

1a.y: Grammar rules (2)

```
term    : term '*' factor          // Just more of the same
          {
            $$ = $1 * $3;
          }
        | term '/' factor
          {
            $$ = $1 / $3;
          }
        | factor
          {
            $$ = $1;
          };

factor  : '(' expr ')'
          {
            $$ = $2;
          }
        | NUMBER
          {
            $$ = $1;
          };
```

1a.y: C code

```
// Global to hold result
double  result      = 0.0;

// Globals to hold input
char*   textPtr     = NULL;
char*   textEndPtr  = NULL;

// Fnc to print error msgs
int yyerror (char *cPtr)
{
  printf("%s, sorry!\n",
         cPtr);
  return(0);
}
```

```
// Call our parser
int     main    (int argc, char* argv[])
{
  char  line[LINE_LEN];

  if  (argc >= 2)
    textPtr = argv[1];
  else
  {
    printf("Expression: ");
    textPtr = fgets(line,LINE_LEN,stdin);
  }

  textEndPtr = textPtr+strlen(textPtr);
  yyparse();
  printf("%g\n",result);
  return(EXIT_SUCCESS);
}
```

Okay, let's understand

- Recall, flex makes a C-function called **yylex()**
  - Returns an integer corresponding to token id
- Helper functions/macros/variables:
  - **YYINPUT()**: Get a bunch of chars of input
  - **yywrap()**: Returns 0 until there are no more chars to get
  - **yylval**: Holds value associated with current token (values held by $1, $2, etc.)
- YACC/Bison, makes **yyparse()**
  - Parses input
- Helper functions/macros:
  - **yylex()**: Gets integer of next token (Hey! Haven't I seen that somewhere?)
  - **yyerror()**: Prints error messages

Question 1: So, how does it know the type of yylval, $1, $2, etc.?

- Answer: **YYSTYPE**

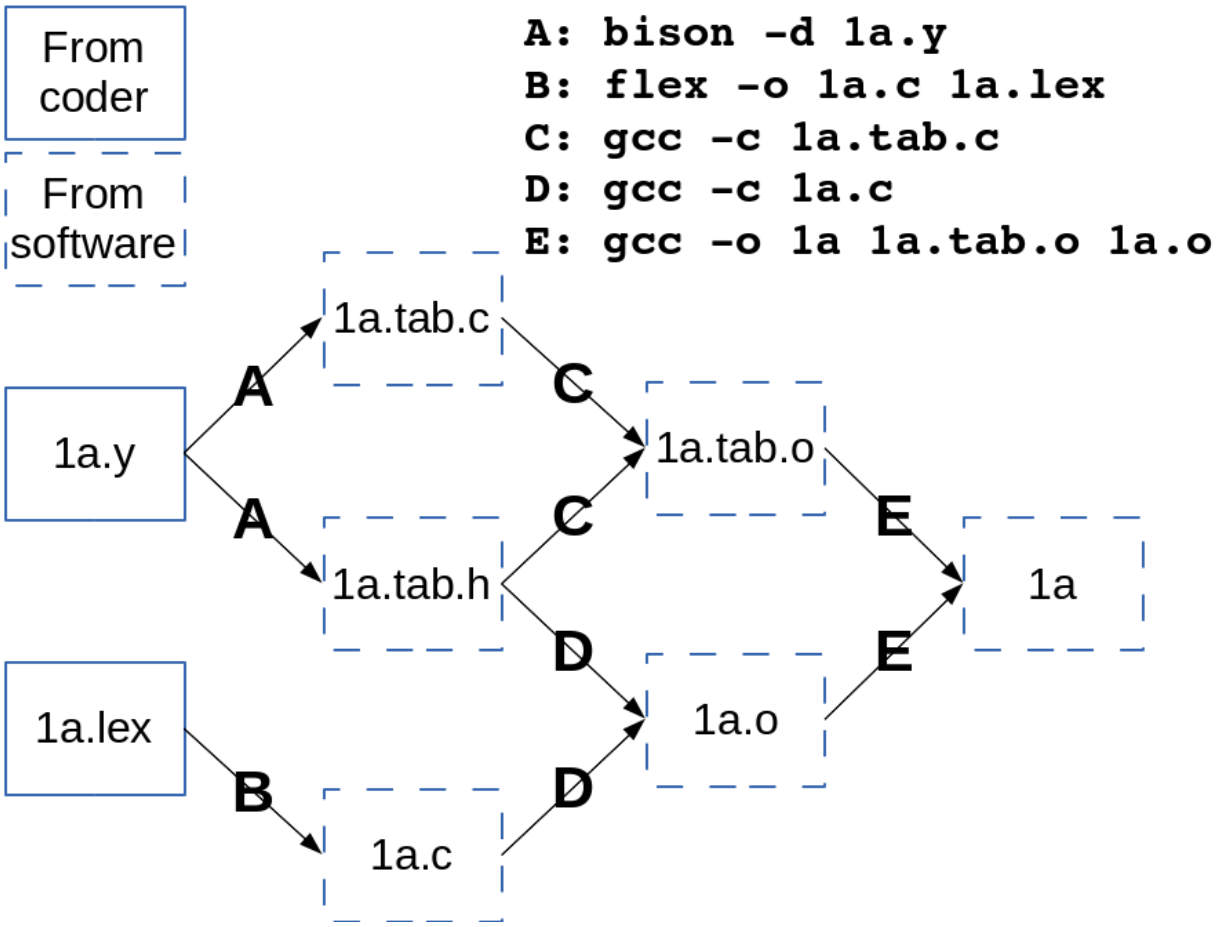
```
//  1a.h
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define  LINE_LEN  256
#define  YYSTYPE   double
extern   double    result;
extern   char*     textPtr;
extern   char*     textEndPtr;
extern   int       yyerror  (char *s);
extern   int       yylex    ();
```

Question 2: 1a.y declared NUMBER and ERROR, how to tell 1a.lex?

- **Answer: Via 1a.tab.h**:
  - Created by Bison for other C files to import

```
. . .
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other
debuggers know about them.  */
   enum yytokentype {
     NUMBER = 258,
     ERROR = 259
   };
#endif
. . .
```

So this, call sequence is complicated . . .



So there is software to manage making it

```
# Makefile for 1a
1a : 1a.tab.o 1a.o
  gcc -o $@ 1a.tab.o 1a.o

1a.o  : 1a.c 1a.tab.h
  gcc -c 1a.c

1a.tab.o : 1a.tab.c 1a.tab.h
  gcc -c 1a.tab.c

1a.c  : 1a.lex
  flex -o $@ 1a.lex

1a.tab.c : 1a.y
  bison -d 1a.y --debug --verbose

1a.tab.h : 1a.y
  bison -d 1a.y --debug --verbose
```

```
$ rm 1a *.o 1a.c *.tab.*
$ make
bison -d 1a.y --debug
gcc -c 1a.tab.c
flex -o 1a.c 1a.lex
gcc -c 1a.c
gcc -o 1a 1a.tab.o 1a.o
$ ./1a
Expression: 5 * 7 + 2
37
```

And what do all those command line options mean?

`bison -d 1a.y --debug --verbose`

- **-d**: Create header file **`1a.tab.h`**
- **--debug**: Add C symbol names to make it easier to use debugger in **yyparse()**
- **--verbose**: Create file **`1a.output`**, which tells grammar states, and shift/reduce/accept rules
  - Very useful when want to debug an ambiguous grammar
- **-p myName**
  - Call everything `myNameparse()`, `myNamelval`, *etc*. instead of `yyparse()`, `yylval`, *etc*.

Getting YACC/Bison to handle precedence & associativity for us

- Old:

```
%start      expr
%nonassoc   '+' '-' '*' '/' '(' ')'
NUMBER
%nonassoc   ERROR

%%

expr : expr ‘+’ term
     | expr ‘-’ term
     | term;
term : term ‘*’ fact
     | term ‘/’ fact
     | fact;
fact : ‘(‘ expr ‘)’
     | NUMBER;
```

- New:

```
// %left %right: associativity
// Ordering (first=lowest,
last=highest): precedence
%left   PLUS MINUS
%left   STAR SLASH
%token BEGIN_P END_P NUMBER
%token ERROR

%%

//  Simpler grammar
expr    : expr PLUS expr
        | expr MINUS expr
        | expr STAR expr
        | expr SLASH expr
        | BEGIN_P expr END_P
        | NUMBER;
```

1b.y rules

```
expr : expr PLUS expr { result = $$ = $1+$3; }
  | expr MINUS expr    { result = $$ = $1-$3; }
  | expr STAR expr     { result = $$ = $1*$3; }
  | expr SLASH expr    { result = $$ = $1/$3; }
  | BEGIN_P expr END_P { result = $$ = $2; }
  | NUMBER             { result = $$ = $1; };
```

Definitions of tokens

- **Associativity**:
  - **`%token`**
    - Just a terminal
    - E.g. constants
    - Not for operators with associativity, precedence
  - **`%left`**, **`%right`**
    - For operators to define left (or right) associativity
  - **`%nonassoc`**
    - For operators with neither associativity
- **Precedence**:
  - Tokens ordered from lowest precedence to highest

A usage of precedence:

- Our grammar has no unary negation:

```
$ ./1b
Please enter an expression: 4 - - 5
syntax error, sorry!
4
```

A New! Improved! Grammar Now with unary negation precedence!

```
%start      expr
%left       PLUS MINUS
%left       STAR SLASH
%nonassoc UMINUS
%token      BEGIN_P END_P NUMBER
%nonassoc   ERROR

/* UMINUS is not a token to
parse, but a precedence */
```

The change in grammar rules

```
expr     : expr PLUS expr
           {
             result = $$ = $1 + $3;
           }
...
         | MINUS expr %prec UMINUS
           {
             result = $$ = -$2;
           }
...
/* UMINUS is not a token to parse, but a
precedence */
```

Et voilà!

```
$ ./1c
Please enter an expression: 4 - - 5
9
```

Hey! Wouldn't it be nice to have ***variables*** too?

- We need some way to store them:
- We'll use C++ std::map<std::string,double>
- Don't know C++? Don't freak out! Here's the interface:

```
class  VarStore
{
public :
  void assign (const char* varNamePtr,
               double value);
  double retrieve(const char* varNamePtr);
};
```

Typing Tokens

```
%union
{
  double      value_;
  char*       charPtr_;
}
```

```
%start      list
%token      PRINT
%right      EQUAL
%left       PLUS MINUS
%left       STAR SLASH
%nonassoc   UMINUS
%token      BEGIN_P END_P END
%token   <charPtr_>   VARIABLE
%token   <value_>     NUMBER
%type    <value_>     expr
%type    <value_>     state
%nonassoc             ERROR
```

Our grammar

```
list    : list state
          {
          }
        |
          {
            // lambda production
          };

state   : PRINT expr END
          {
            printf("%g\n",$2);
          }
        | expr END
          {
            $$ = $1;
          };
```

Our grammar

```
expr    : VARIABLE EQUAL expr
          {
            varStore.assign($1,$3);
            $$ = varStore.retrieve($1);
          }
        | expr PLUS expr
          {
            $$ = $1 + $3;
          }
        | expr MINUS expr
          {
            $$ = $1 - $3;
          }
        | expr STAR expr
          {
            $$ = $1 * $3;
          }
        | expr SLASH expr
          {
            $$ = $1 / $3;
          }
        | MINUS expr  %prec UMINUS
          {
            $$ = -$2;
          }
        | BEGIN_P expr END_P
          {
            $$ = $2;
          }
        | NUMBER
          {
            $$ = $1;
          }
        | VARIABLE
          {
            $$ = varStore.retrieve($1);
          };
```

Our new tokenizer:

```
[ \t\n]   { /* ignore spaces */ }
[0-9]+|([0-9]*\.[0-9]+) {
     yylval.value_ =
         strtod(yytext,NULL);
     return(NUMBER);
                         }
print     { return(PRINT); }
[a-zA-Z_][a-zA-Z_0-9]*  {
     yylval.charPtr_ =
         strdup(yytext);
     return(VARIABLE);
                         }
```

```
\+       { return(PLUS); }
\-       { return(MINUS); }
\*       { return(STAR); }
\/       { return(SLASH); }
\(       { return(BEGIN_P); }
\)       { return(END_P); }
=        { return(EQUAL); }
;        { return(END); }
.        {
  printf("What's '%c'?\n",
          yytext[0]);
  return(ERROR);
         }
```

Other code:

```
// ld.h
. . .
extern  VarStore  varStore;
extern double     result;

// ld.y
. . .
VarStore          varStore;
double            result;
```

Does she work?

```
$ ./1d
Please enter an expression: var1 = 17; var2 = 0.5; var3 = var1 * var2 + 3; print var1; print var2; print var3;
17
0.5
11.5
```

Awesome! Is there anything YACC/Bison can ***not*** do?

- Remember: it is LALR(1).
- It cannot handle grammars if has insufficient lookahead:

```
phrase : cartAnimal AND CART
       | workAnimal AND PLOW;
cartAnimal : HORSE | GOAT;
workAnimal : HORSE | OX;
```