* In Simple.lex file:
"const" {return CONST;}

* In Simple.y:
Add this entry in the %union directive

```
struct {
      int type;
      int value;
} expval;
```

and add these declarations:

```
%token CONST
%type <expval> exp
```

You must also modify the symbol table, adding two integer fields:
"type", which will be set to 0 for variables and to 1 for constants; and
"value", which will be set to the constant value for constants, and 0
otherwise.

* In the syntax part:

Constants are declared after variable declarations; the new non-terminals
"named_const" and "named_const_list" are used for the purpose:

```
program
   : DECLARATIONS declarations named_const BEGIN_PROGRAM commands END_PROGRAM
   ;

named_const
   : /* empty */
   |  CONST INTEGER named_const_list IDENTIFIER '=' NUMBER '.'
   ;

named_const_list
   : /* empty */
   | named_const_list IDENTIFIER '=' NUMBER ','
   ;
```

* In the semantic part:

Extend the functions "insert" and "putsym" so as they can receive two more
arguments: "type" and "value". "type" is a Boolean argument, 0 for variables and
1 for constants. "value" is used for constants only.

For unary expressions, if the subexpression is constant,
propagate its value in "$$.value" and set "$$.type" to 1, otherwise set it
to 0 and do nothing for "$$.value". If the expression is constant, do not
generate code. For example:

```
exp : IDENTIFIER
     {
        symrec *identifier;
        identifier = getsym( $1 );
        if ( identifier == 0 ) {
           errors++;
           printf( "%s is an undeclared identifier\n", $1 );
        } else {
           if (identifier->type == 0) {
             gen_code( LD_VAR, identifier->offset );
```

```
            $$.type = 0;
          } else {
            $$.value = identifier->value;
            $$.type = 1;
          }
       }
     }
```

For binary expressions, if both subexpressions are constant, propagate the result value and set "$$.type" to 1; do not generate code. If both are variable, everything stays as in the original version, plus "$$.type" is set to 0. Otherwise, generate LD_INT for the constant subexpression before generating the appropriate operation code as in the standard version. For example:

```
exp : exp '+' exp
     {
       if ($1.type == 1 && $3.type == 1) {
          $$.value = ($1.value + $3.value);
          $$.type = 1;
       } else {
         if ($1.type == 0 && $3.type == 1)
            gen_code( LD_INT, $3.value );
         else if ($1.type == 1 && $3.type == 0) {
            gen_code( LD_INT, $1.value );
         }
         gen_code( ADD, 0 );
         $$.type = 0;
       }
     }
```

Each time the "exp" non-terminal is used in a command, if the expression is constant a LD_INT with the expression value must be generated. For example:

```
command : WRITE exp
     { if ($2.type == 1) gen_code( LD_INT, $2.value );
       gen_code(WRITE_INT, 0 );}
```

* Bonus: there is an additional issue that should be taken into account with non-commutative operations (expecially divisions): if the first operand is constant and the second is variable, this can only be detected after code for the latter has been generated, while no code for the first has been generate yet. The solution is to preserve the order of operands by swapping them. This requires either a SWAP bytecode or two temporary variables. For example, introducing a SWAP opcode:

```
exp : exp '/' exp
     {
       if ($1.type == 1 && $3.type == 1) {
          $$.value = ($1.value / $3.value);
          $$.type = 1;
       } else {
         if ($1.type == 0 && $3.type == 1)
            gen_code( LD_INT, $3.value );
         else if ($1.type == 1 && $3.type == 0) {
            gen_code( LD_INT, $1.value );
            gen_code( SWAP, 0 );
         }
         gen_code( DIV, 0 );
         $$.type = 0;
       }
     }
```

The SWAP opcode is interpreted in "SM.c":

```
    case SWAP : { int t = stack[top];
                  stack[top] = stack[top-1];
                  stack[top-1] = t;
                  break; }
```