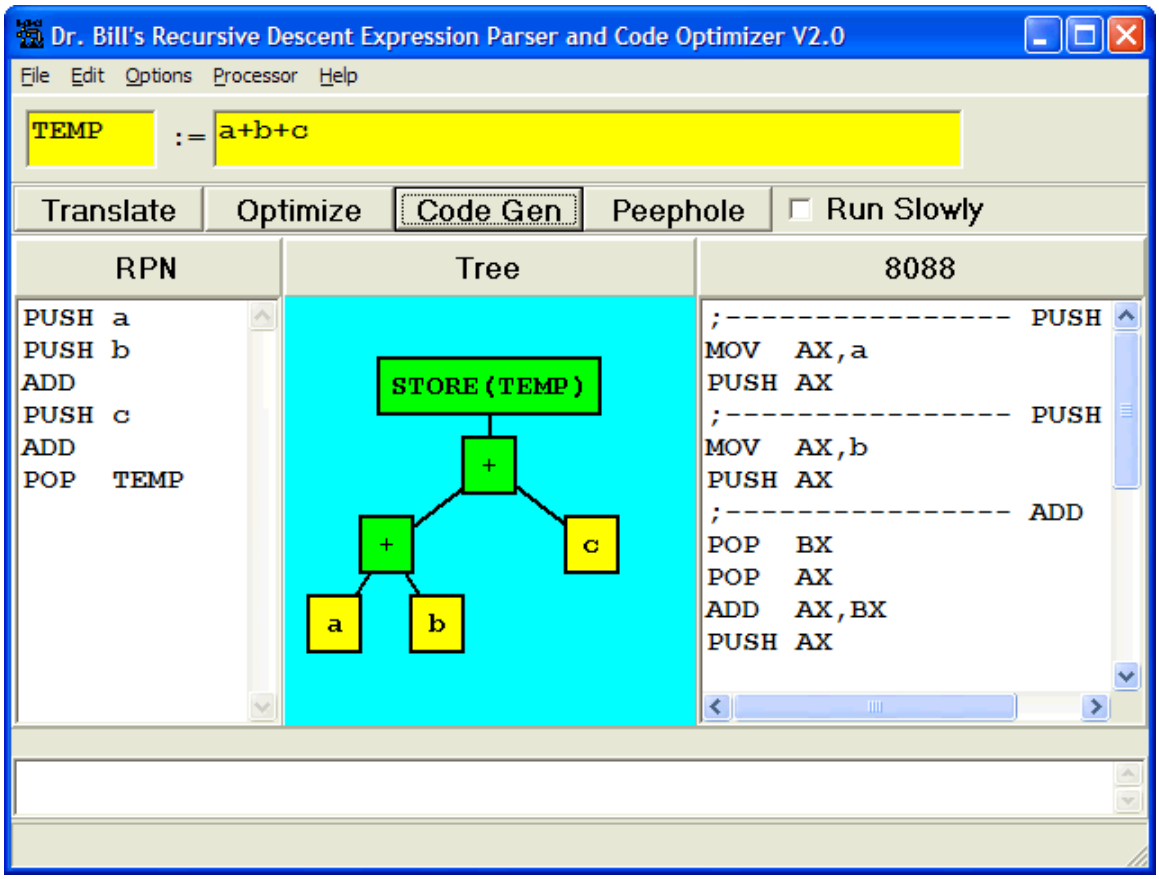
# Manual for

## *"Dr. Bill's Expression Parser and Code Optimizer"*



**Dr. William T. Verts**





## Introduction

This program was designed as a classroom aid to show students how arithmetic expressions are translated and optimized by a high-level language compiler. The student types in an arithmetic expression into an edit box, and then clicks on a button that translate the statement into Reverse Polish Notation (RPN) and a graphical binary tree format. Another button performs optimization steps on the binary tree, and regenerates the resulting RPN. A third button translates the existing tree into one of three assembly language formats. The fourth and final button performs "peephole" optimization on the assembly language in order to reduce the number of total instructions.

Students may try many different expressions to see how each one is compiled and optimized. They may also choose whether or not to attempt optimization on the tree before assembly code is generated, and they may choose whether or not to attempt optimization of that assembly code.

Observant students will note that in many cases the "best" code is generated when *both* tree optimization and peephole optimization are performed. While both forms of optimization catch many of the same cases, each will optimize cases completely missed by the other approach. Together, they generate pretty good code. Observant students will also notice that the final code can be tightened up even further in some circumstances. This program was not intended to generate "perfect" code, but was instead designed to illustrate many of the common approaches taken by optimizers. This can lead to many discussions in both assembly language and compiler classes as to how to better approach these kinds of problems.

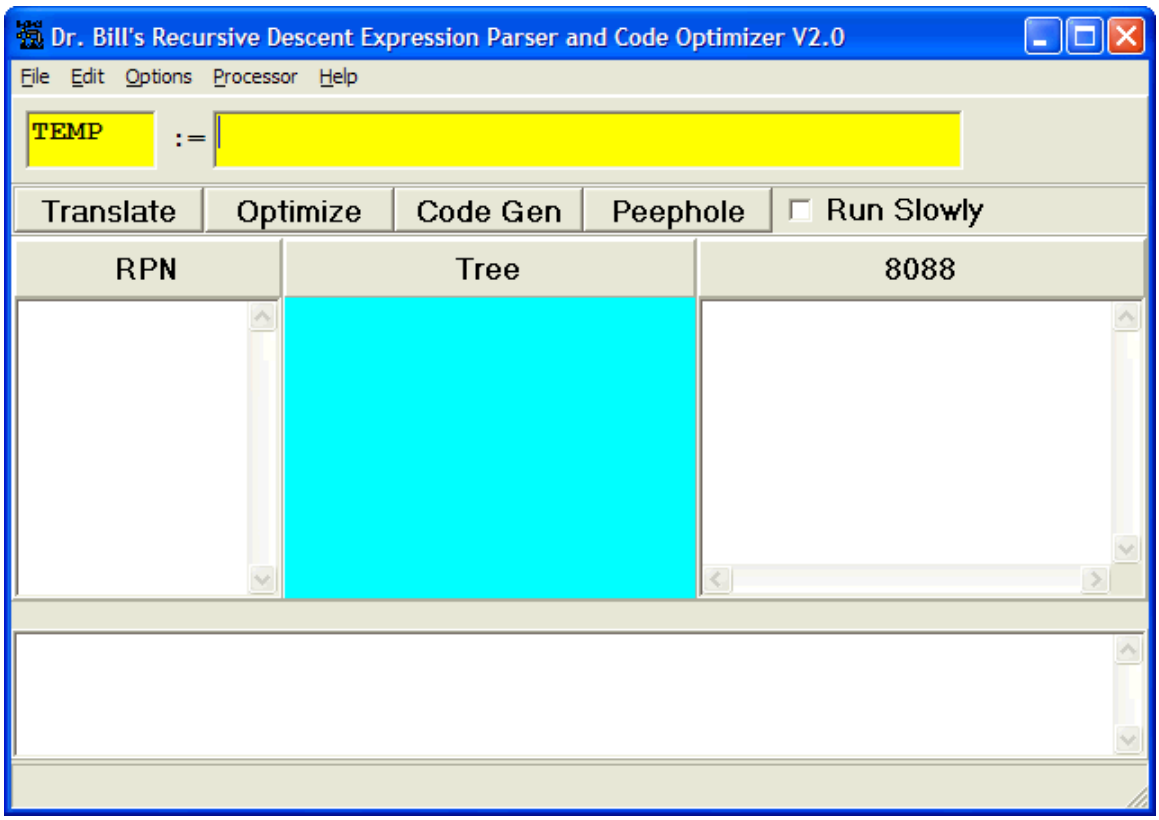
The program was first written for a compiler class that I taught at Smith College in the spring semester of 1999, and this first version produced only 8088 assembly language code. I dusted it off and updated the interface (and corrected a few bugs) for the fall 2002 assembly language class at UMass, Amherst. For the fall 2004 class I added the ability to generate assembly code for the ARM chip; both integer and floating point instructions.

## System Requirements

This program will run on any Intel 486, Pentium, or later computer equipped with Windows 95 or later. While the program will work on screens as small as 640×480, the graphical area in the center will be too small for all but the simplest expression trees to show completely. A screen at least 800×600 is recommended (1024×768 or 1280×1024 will give the best results). The program does not require any installation or special folder; as long as the `.EXE` file has been unpacked from its archive, it will run from any location (including from floppy disks and CD-ROMs). This `.PDF` document should be retained in the same directory as the `.EXE` file so it can be launched from the program with the F1 key.

## Opening Screen Geometry

Upon launch, the Expressions program will be a maximized version of the following view. The window may be scaled to any appropriate size. It is recommended that the window be left in its original, maximized state.




The user types his or her arithmetic expressions into the yellow edit box just under the menu bar. By default the result of the expression is stored into a variable called `TEMP`, but the user may change this variable name as well.

By clicking the Translate button, the expression is parsed and converted into RPN, which appears in the "RPN" panel. The RPN is also converted into a parse tree, shown graphically in the "Tree" panel. Any errors in the syntax of the expression are detected at translation time; if any are present an error dialog appears and the RPN and Tree windows remain empty.

Clicking the Optimize button causes the program to examine the parse tree in an attempt to make it more efficient. Depending on the expression, the parse tree may be greatly altered or it may not change at all. Any optimizations performed are logged in the wide panel at the bottom of the screen.

Clicking the Code Gen button generates working assembly code from the current parse tree, which is then shown in the rightmost panel.

Clicking the Peephole button optimizes the current assembly code in place. Any optimizations performed are logged in the wide panel at the bottom of the screen.

The Run Slowly check box controls the speed of the optimization steps; if checked there is about a ½ second delay after each optimization is logged before the next step is performed. If the check box is not checked, the optimization steps run at full speed. This check box has no effect on either the Translate or the Code Gen steps.

## Menu Commands

There are very few menu commands, few of which greatly affect the on-screen operation of the program. The menu items are ass follows:

File-Save Image... (Ctrl S)

Brings up a dialog box to save the current diagram in the "Tree" panel to a 16-color `.BMP` file.

File-Exit

Exits the program. There are no options to save any of the expressions or panel contents, so there will be no confirmation dialogs. The program will be terminated.

Edit-Copy RPN to Clipboard

Copies the text contents of the RPN panel to the clipboard. If a portion is selected, only that portion will be copied. If no text in that panel has been selected, the entire contents will be copied.

Edit-Copy Assembly to Clipboard

Copies the text contents of the rightmost assembly language panel to the clipboard. If a portion is selected, only that portion will be copied. If no text in that panel has been selected, the entire contents will be copied.

Edit-Copy Optimization List to Clipboard

Copies the text contents of the bottom panel (where any optimizations are logged) to the clipboard. If a portion is selected, only that portion will be copied. If no text in that panel has been selected, the entire contents will be copied.

Edit-Copy Tree Image to Clipboard (Ctrl C)

Copies the graphical image of the current parse tree to the clipboard. From there it could be pasted into a program such as Windows Paint for annotation.

Options-Show Hints (checked on by default)

By default, floating the mouse over any panel or control in the program will show a brief description of the purpose of that object. This menu item toggles that action on and off.

Options-Increase Tree Depth (Ctrl I)

This increases the distance between tree levels in the graphical image, making the tree appear to be deeper. Adjust this setting to get the graphical tree "looking just right" for a particular expression.

Options-Decrease Tree Depth (Ctrl D)

This decreases the distance between tree levels in the graphical image, making the tree appear to be shallower. Adjust this setting to get the graphical tree "looking just right" for a particular expression.

Processor-8088

This sets the assembly language mode to produce code for the Intel-8088 when the Code Gen button is clicked. This is the default mode when the program starts.

Processor-ARM (integer)

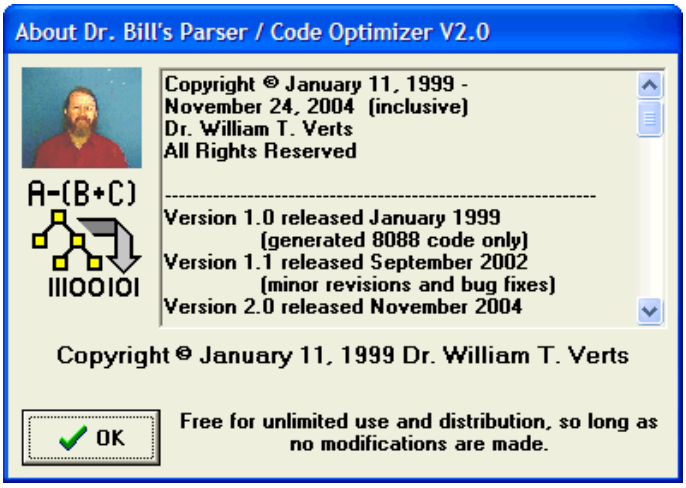
This sets the assembly language mode to produce integer code for the ARM RISC chip when the Code Gen button is clicked.

Processor-ARM (floating pt.)

This sets the assembly language mode to produce floating point code for the ARM RISC chip when the Code Gen button is clicked.

Help-About…

This brings up the standard "About" box, showing copyright and basic information about the program:




Help-Manual… (F1)

This launches Adobe® Acrobat Reader (if installed) and loads the current document that you are reading now.

## Entering Arithmetic Expressions

Expressions are all of the form ***variable := expression***, where the default assignment variable is called `TEMP`. This variable may be renamed, but doing so does not change the resulting generated code (even though, perhaps, it should in a few cases). The assignment operator is part of the demonstration program and may not be changed.

Expressions are typed into the edit box just as if a programmer was to write those statements in a programming language such as Pascal. The standard four mathematical operators for add, subtract, multiply, and divide are supported.

Normal precedence rules are in force ("*" and "/" before "+" and "-"), and parentheses are used to override the natural precedence of the operators.

Unary minus and unary plus are *not* supported; neither are common intrinsic functions such as ABS, SQRT, etc.

Variables are considered to be any string starting with a letter and containing letters and the underscore but *not* digits. For example, variables `A`, `A_B_C`, and `Temp` are all legal, but `A1` and `T34X` are not legal. Use of variable names that mimic assembly language register names is not recommended, as this may confuse the peephole optimizer (for example, do not use variables named `AX` or `BX` when producing 8088 code or `SP` when producing ARM code).

Constants are integers only, consisting of the digits 0 through 9. No floating-point numbers are allowed. Because unary minus is not supported, negative constants are not allowed either.

These restrictions make the parser easier to construct and debug, and do not greatly affect the operation of the program. These restrictions may be lifted in a future release/rewrite of the program.

Conditionals are also allowed, such as `(T > 0)` as part of an expression. The semantics of this construct are that "true" will be treated as a number equal to 1, and "false" will be treated as a number equal to 0.

The following example expressions are all legal:

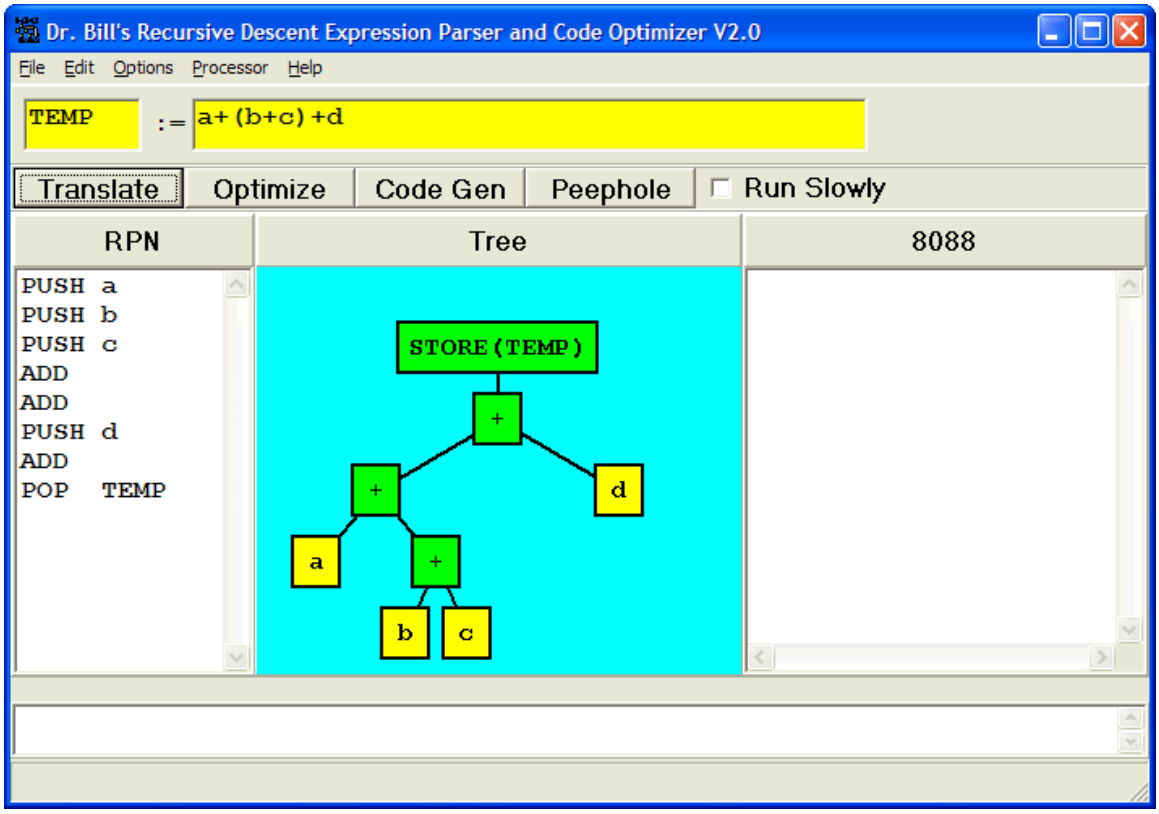
```
TEMP := a+b+c+d
TEMP := a+(b+c)+d
TEMP := (a+(b+(c+d)))
TEMP := a+(b*c)-(d>0)
TEMP := a*2+(3-5)+c
```

## Translating Expressions into RPN

Clicking the Translate button translates the current expression into RPN and generates the corresponding tree. For example, translating the expression

```
TEMP := a+(b+c)+d
```

will result in the following image:

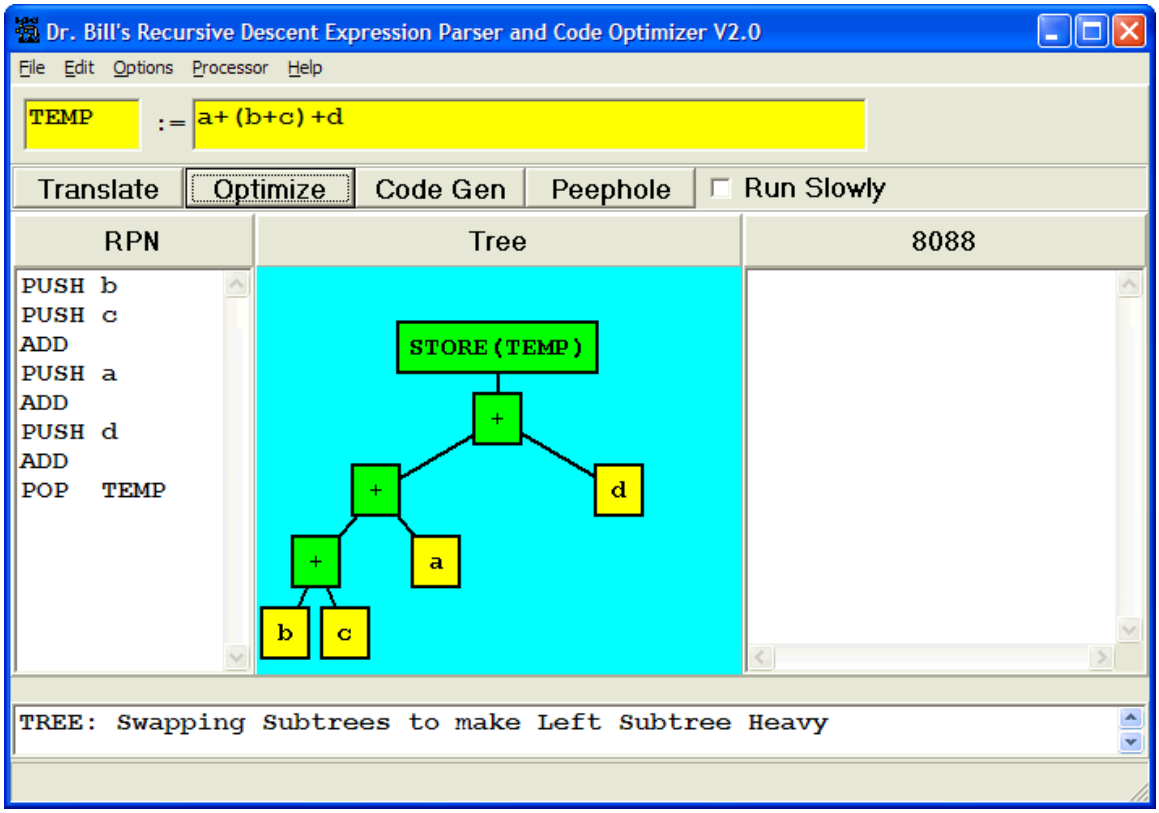



Changing the expression will not change either the RPN or the Tree views until the Translate button is clicked again.

The RPN simulates a generic "stack machine" where items (both variables and constants) are placed onto the stack with PUSH operations. Arithmetic operators pop two items off of the stack, do their things, and then push the result back onto the stack. The final POP operation removes the single remaining item from the stack and stores its value into the specified memory variable.

## Optimizing the Tree

Code can be generated from the existing tree (skipping this step), or by clicking the Optimize button the program will attempt to create a functionally equivalent tree with fewer nodes or nodes arranged in a manner more conducive to good code generation. These steps include constant folding (mapping `(3+4)` onto `7`, for example), dead code elimination (replacing `(a-a)` terms with `0`, for example), and swapping subtrees for "+" and "*" operators to make the left subtree heavier (deeper) than the right subtree. This last step tends to reduce extraneous pushes and pops of the stack, and was the single rule used on the tree from the previous page to obtain the following "optimized" tree:




Compare the RPN code in this version with that of the image on the previous page. In the earlier version, the stack had three items on it at the same time after the three initial PUSH operations. In this optimized version, the stack never contains more than two items at any one time. Fewer items on the stack will tend to generate fewer assembly language statements at the end.

## Generating Code

By clicking the Code Gen button, the program creates assembly language by mechanically converting each RPN instruction into the equivalent code for whichever processor model has been selected. By default, the mode at program start-up is set to generate code for the Intel 8088.

***Processor 8088***

In this mode the program creates 16-bit 8088 assembly language. The 16-bit AX register of the 8088 is used for most of the arithmetic operations.

Each PUSH instruction in the RPN is translated into a MOV AX,xxx where the xxx is either a variable name or a 16-bit integer constant, followed by a PUSH AX instruction. For example, PUSH a is translated into

```
MOV  AX,a
PUSH AX
```

Each ADD instruction in the RPN generates a pop into BX, a pop into AX, an ADD of BX into AX, and a push of AX. Subtraction works in the same manner, as does multiplication (which always uses AX, so it need not be specified in the instruction). You should be able to understand now why the first operand on the stack is always popped into BX instead of into AX.

```
POP  BX        POP  BX        POP  BX
POP  AX        POP  AX        POP  AX
ADD  AX,BX     SUB  AX,BX     IMUL BX
PUSH AX        PUSH AX        PUSH AX
```

Division is similar, but one extra instruction is needed because of the peculiarity of the 8088 chip. Integer division always divides a 32-bit operand in DX:AX by the 16-bit divisor, so the value in AX must be sign-extended into DX before the division can take place. Sign extension is done with the CWD instruction (convert word to double word), as in:

```
POP  BX
POP  AX
CWD
IDIV BX
PUSH AX
```

Storage of the computed value into the result variable requires the following code (a pop off of the stack):

```
POP  AX
MOV  TEMP,AX
```

Code generation for conditional expressions requires the creation of an If-Then-Else, using two labels, one conditional jump instruction, and one unconditional jump. The resulting code is fairly complicated, and is difficult to optimize using simple techniques, which is why it was included in this program. For example, the conditional expression `(a>b)` is assembled as follows:

```
POP  BX
POP  AX
CMP  AX,BX
JG   L0001
MOV  AX,0
JMP  SHORT L0002
L0001:
MOV  AX,1
L0002:
PUSH AX
```

The labels `L0001` and `L0002` in the example above are automatically and uniquely generated by the program, so that in cases where there are several such conditional expressions no conflicts between labels can arise.

For the example given earlier of `TEMP := a+(b+c)+d`, the unoptimized tree gives the following 8088 code (22 assembly language statements with comments between each virtual RPN statement). Study the code carefully to make certain you understand what each step is accomplishing.

```
;---------------------- PUSH a
MOV  AX,a
PUSH AX
;---------------------- PUSH b
MOV  AX,b
PUSH AX
;---------------------- PUSH c
MOV  AX,c
PUSH AX
;---------------------- ADD
POP  BX
POP  AX
ADD  AX,BX
```

```
PUSH AX
;--------------------- ADD
POP  BX
POP  AX
ADD  AX,BX
PUSH AX
;--------------------- PUSH d
MOV  AX,d
PUSH AX
;--------------------- ADD
POP  BX
POP  AX
ADD  AX,BX
PUSH AX
;--------------------- POP TEMP
POP  AX
MOV  TEMP,AX
```

For the optimized tree the code is as follows (also 22 assembly language statements, but rearranged so the virtual RPN "stack" is never larger than necessary):

```
;--------------------- PUSH b
MOV  AX,b
PUSH AX
;--------------------- PUSH c
MOV  AX,c
PUSH AX
;--------------------- ADD
POP  BX
POP  AX
ADD  AX,BX
PUSH AX
;--------------------- PUSH a
MOV  AX,a
PUSH AX
;--------------------- ADD
POP  BX
POP  AX
ADD  AX,BX
PUSH AX
;--------------------- PUSH d
MOV  AX,d
PUSH AX
;--------------------- ADD
POP  BX
POP  AX
```

```
ADD  AX,BX
PUSH AX
;--------------------- POP TEMP
POP  AX
MOV  TEMP,AX
```

If you examine the 8088 code carefully, you will notice that while the code will work it pushes and pops items from the 8088 stack far more frequently than it ought to. In particular, there are several cases where a `PUSH AX` instruction is followed directly by a `POP BX` instruction. Since these 8088 instructions were in *separate* RPN virtual instructions, the code generator didn't recognize that they could have been replaced by a single `MOV BX,AX` instruction (which doesn't use the 8088 stack at all). Thus, one optimization rule is "replace all `PUSH AX | POP BX` instruction pairs by the single instruction `MOV BX,AX`".

Clicking the Peephole button starts the optimization process at the 8088 code level. In the optimizer there are a large number of similar rules, all of which look for patterns of instructions in the code that can be replaced by fewer (or simpler) instructions. Since these patterns of instructions tend to be fairly close together, optimization is said happen by "looking through a small peephole" at the code. The optimizer is making very local changes to the code, instead of looking at the higher-level semantics (such as would be done by the tree optimizer). The ability of the peephole optimizer to function is controlled by the number of rules it has available; in general the more rules it has the better code it can generate, but the longer it will take to perform the optimization.

Here is that first optimization rule again:

```
PUSH AX         replaced by   MOV  BX,AX
POP  BX
```

Once those redundant PUSH/POP instructions are replaced by a `MOV`, the next pattern that shows up is a `MOV AX,xxx` instruction followed by a `MOV BX,AX` instruction. This pattern can be replaced by a single `MOV BX,xxx` instruction:

```
MOV  AX,xxx     replaced by   MOV  BX,xxx
MOV  BX,AX
```

After a few such transformations, another interesting pattern emerges:

```
PUSH AX
{ instructions that don't use AX at all }
POP  AX
```

In this case both the `PUSH` and the `POP` instructions can be removed because they are saving and restoring a register that isn't modified in the interim.

Some other examples of rules used in this program are as follows:

Replace a multiply by 2 by a left-shift (strength reduction):

```
MOV  BX,2       replaced by   SHL  AX,1
IMUL BX
```

Replace a three-byte `MOV` of the special constant zero with a single byte `XOR` statement:

```
MOV  AX,0       replaced by   XOR  AX,AX
```

Replace a `MOV` of a constant followed by an `ADD` or `SUB` of a constant with a `MOV` of the correct value (constant folding):

```
MOV  AX,4       replaced by   MOV  AX,7
ADD  AX,3
```

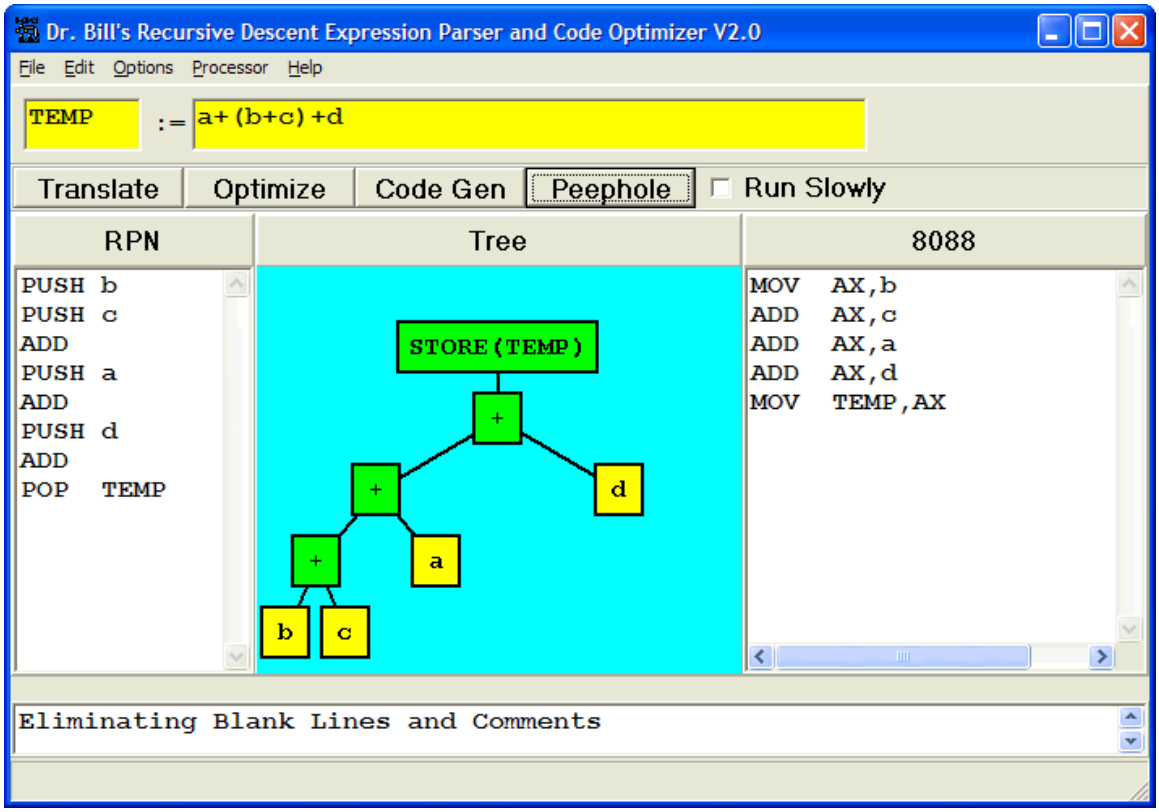
Applying peephole optimization to the 8088 code generated by the unoptimized tree for `TEMP := a+(b+c)+d` gives the following result (9 statements instead of 22). Notice that variable `a` is pushed onto the stack much earlier than it needs to be, but the peephole optimizer can't fix the problem:

```
MOV  AX,a
PUSH AX
MOV  AX,b
ADD  AX,c
MOV  BX,AX
POP  AX
ADD  AX,BX
ADD  AX,d
MOV  TEMP,AX
```

Applying the same peephole optimization rules to the 8088 code generated by the optimized tree for `TEMP := a+(b+c)+d` gives the following result (only 5 statements remain of the original 22, about as good as you can get):

```
MOV  AX,b
ADD  AX,c
ADD  AX,a
ADD  AX,d
MOV  TEMP,AX
```

Here is how the screen will look after the entire process is complete. The tree has been optimized, the 8088 code generated, and the 8088 code has been peephole optimized. The final result in the rightmost panel is what a compiler should actually generate.




***Processor ARM (both integer and floating point)***

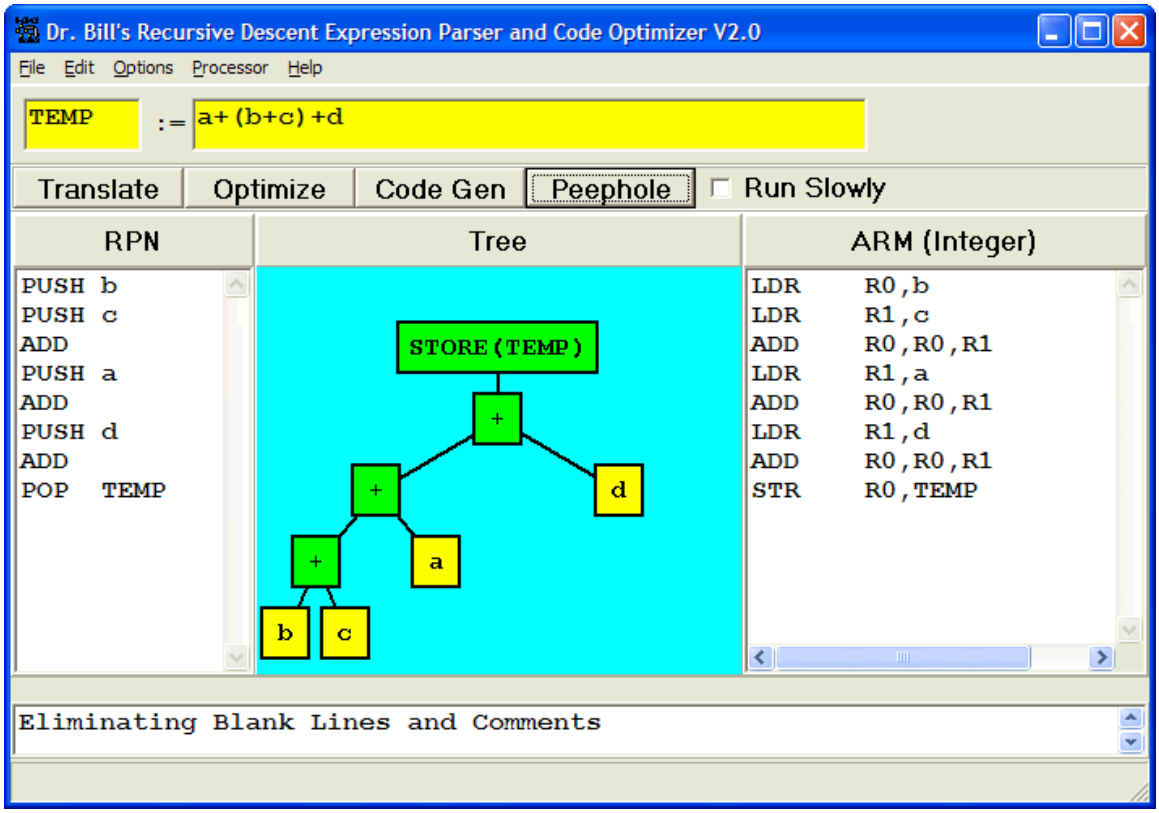
The process for optimizing code generated for the ARM is very similar to that for the Intel 8088. Each RPN instruction is converted mechanically into an equivalent set of ARM instructions. On the ARM there are sixteen 32-bit integer registers, `R0` through `R15` (although `R12` through `R15` have specific uses). In the code generated here, `R0` and `R1` are the only registers ever used; an advanced compiler may make use of more registers to further reduce the number of instructions over larger blocks of code. There are also eight floating-point registers, `F0` through `F7`, which are also 32 bits in length when treated as single precision (double precision is not currently supported by this program).

Since the ARM is a three-address machine instead of a two-address machine such as the 8088, every instruction except for simple moves will contain *two* source operands and a destination. The pattern matcher looks for specific registers used by certain

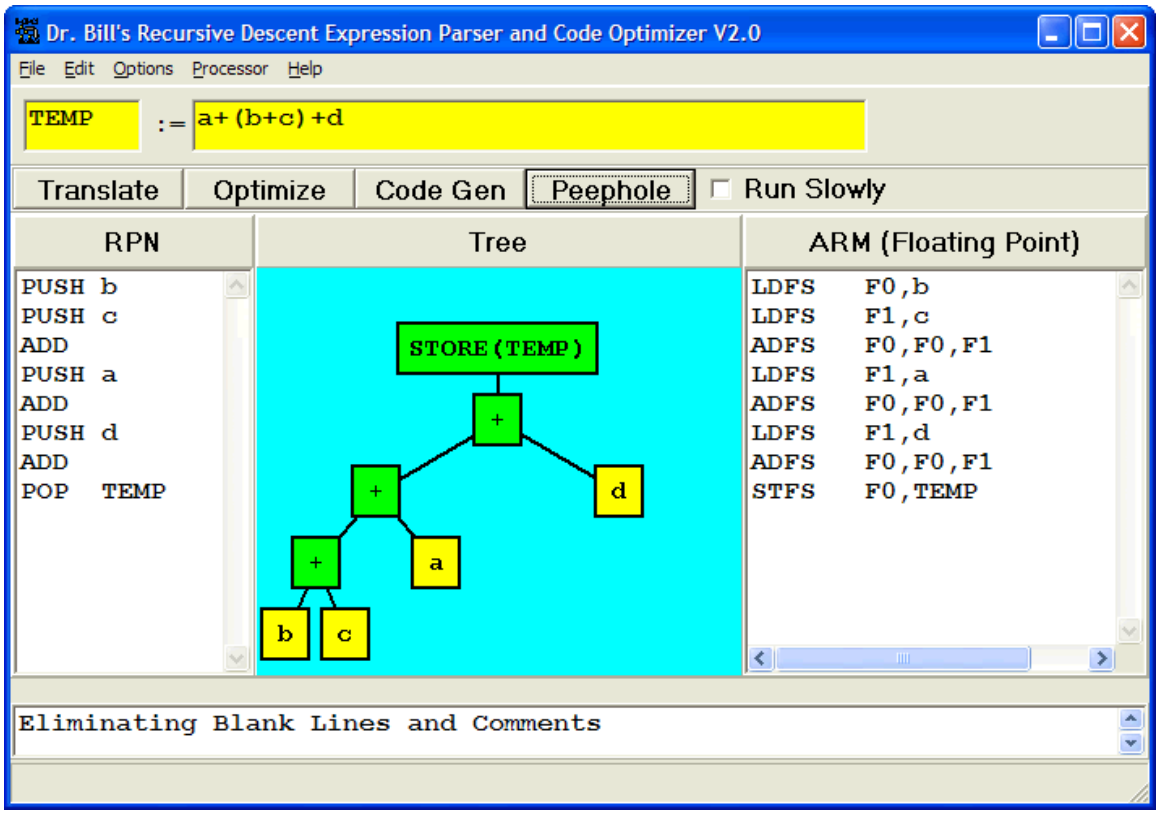
instructions in order to determine whether or not a sequence of instructions can be optimized.

In the 8088, a **PUSH a** pseudoinstruction is translated into `MOV AX,a` followed by `PUSH AX`. On the ARM the same pseudoinstruction becomes `LDR R0,a` followed by `STR R0,[SP,#-4]!` in integer mode. In floating point mode this becomes the instruction `LDFS F0,a` followed by a `STFS F0,[SP,#-4]!` instruction. All three sequences mean exactly the same thing. Every other pseudoinstruction maps onto an equivalent code sequence.

Here is the result for optimizing ARM integer code:



Here is the same result for ARM floating point code:




## Conclusions

As you can see from these simple examples, expressions can be easily parsed into a binary tree form, optimized at that level, mechanically translated into assembly code, and the result peephole optimized.

Compilers designed to create working code as quickly as possible might not optimize at all (i.e., they spend minimal time in the translator but produce terrible code). Compilers that only perform global tree optimization can reduce out a lot of redundancy in the high-level semantics, but the code generator may still not produce very good code for the particular target processor. Compilers that only perform peephole optimization do pretty well on the code for the target processor, but do not produce as good a final result as if a global tree optimization was performed first.

The lesson should be clear: compilers should optimize wherever they can. Having just one type of optimization is better than nothing, but with several different types of optimizations working together remarkably good code can be produced.

## Examples to Try

Try the following examples under all combinations of optimization: none, tree only, peephole only, both. For each example compare the final results and determine the number of instructions that result. See if you can spot cases where further optimization could result in better code, and come up with a pattern-matching rule that addresses each new case.

1. `TEMP := a+b*c+d`

2. `TEMP := (a+(b-(b*(x/x))))/a`

3. `TEMP := 8+5*3`

4. `TEMP := 1+2+3+4`