Lecture #34 – May 3, 2004

Compiler Issues

A traditional programming language requires that programs be written in some kind of text editor, and then passed through a compiler for syntax validation and translation into executable form. Modern integrated development environments bundle the editor and compiler into the same package, often including integrated debugging along with color syntax highlighting, pop-up method completion, dependency graphs, parameter hints for functions and procedures, etc. Fundamentally, however, the core of the process most often requires that a compiler translate a program's text into an equivalent executable form.

Compilers consist of several software modules. Early compilers, constrained by limited memory, often made multiple passes over their input programs, performing some new translation or optimization task during each pass. In these compilers the modules were largely independent of one another. In modern compilers the modules are much more interdependent on one another, and all generally run together. Some programming languages are designed to partially reduce the load on the compiler; "standard" Pascal (if there is such a thing anymore) enforces a strict define-before-use ordering of symbols in the text of a program, and a simple compiler can translate the program's source code into an equivalent executable in a single pass.

The traditional modules of a compiler are the ***lexical analyzer***, the ***parser***, the ***optimizer***, the ***code generator***, and the ***peephole optimizer***.

The lexical analyzer deals with converting the characters of the input text into easily identifiable ***tokens***. For example, the sequence of characters **3**, **.**, **1**, **4**, **1**, **5**, and **9** must be converted into the equivalent internal floating point number 3.14159, and the sequence of characters **"**, **H**, **E**, **L**, **L**, **O**, and **"** must be converted into the five-character string object HELLO. Similarly, text items that start with letters and contain letters, digits, and the underscore must be treated as identifiers (symbols), with the restriction that certain special identifiers are reserved words and are to be tagged as such. The lexical analyzer strips out white space, line boundaries, and comments, and passes the stream of recognized tokens to the parser. Any illegally formed tokens (such as missing quotes in strings or badly formed numbers) cause errors to be generated at this time.

The parser looks at the stream of tokens to see if they follow the rules of the language, and creates an equivalent internal tree structure of each statement. Often, statements in the original language are translated into an internal stack-based representation. Illegal statements (token streams that are illegal under the rules of the language) are flagged as errors.

The optimizer examines the internal representation to see if a simpler but equivalent structure can be generated. Examples include ***constant folding*** (replacing the constant expression 4+5 with 9), ***reduction in strength*** (replacing `2×X` with `X+X` if

floating point or `X SHL 1` if integer), elimination of identities (replacing `X-X` with `0` or replacing `X×1` with `X`), ***common subexpression elimination*** (in an array expression such as `A[X+4] := A[X+4] + B[X+4]`, precomputing `Y := X+4` and rewriting the expression as `A[Y] := A[Y] + B[Y]`), ***dead code elimination*** (getting rid of entire structures that map onto never-executed code such as `WHILE TRUE=FALSE DO BEGIN … END`), and so on. The application of one rule may open up the structure for the application of another. The big trick is to never make a transformation that changes the semantics of the program.

One the internal structure has been optimized; it is used to generate machine code. For example, the high-level language statement `C := A + B` is converted into the stack operations `PUSH A`, `PUSH B`, `ADD`, `POP C`. This form is easily translated into real machine code for whatever machine is the target.

8088 CODE	**ARM CODE**	**STACK CODE**
`MOV AX,A`	`LDR R0,A`	*PUSH A*
`PUSH AX`	`STR R0,[SP,#-4]!`	
`MOV AX,B`	`LDR R0,B`	*PUSH B*
`PUSH AX`	`STR R0,[SP,#-4]!`	
`POP BX`	`LDR R1,[SP],#4`	*ADD*
`POP AX`	`LDR R0,[SP],#4`	
`ADD AX,BX`	`ADD R0,R0,R1`	
`PUSH AX`	`STR R0,[SP,#-4]!`	
`POP AX`	`LDR R0,[SP],#4`	*POP C*
`MOV C,AX`	`STR R0,C`	

By blindly translating the stack machine code into the assembly language of your choice you can see that the result is very inefficient. In particular, the code at the start of one stack statement may complement the code at the end of the previous stack statement; a `PUSH` followed immediately by the corresponding `POP`, for example.

At this point a set of peephole optimization rules scans the code through a small "peephole" looking for patterns of instructions which can be replaced by shorter sequences or eliminated entirely. The codes below have eliminated `PUSH` *x* | `POP` *x* pairs, and have replaced all `PUSH` *x* | `POP` *y* pairs by *y* ← *x* instructions.

8088 CODE	**ARM CODE**
`MOV AX,A`	`LDR R0,A`
`PUSH AX`	`STR R0,[SP,#-4]!`
`MOV AX,B`	`LDR R0,B`
`MOV BX,AX`	`MOV R1,R0`
`POP AX`	`LDR R0,[SP],#4`
`ADD AX,BX`	`ADD R0,R0,R1`
`MOV C,AX`	`STR R0,C`

Next, in instruction sequences where data move from register to register to register, intermediate register moves can be folded together as follows:

```
8088 CODE          ARM CODE
MOV  AX,A          LDR  R0,A
PUSH AX            STR  R0,[SP,#-4]!
MOV  BX,B          LDR  R1,B
POP  AX            LDR  R0,[SP],#4
ADD  AX,BX         ADD  R0,R0,R1
MOV  C,AX          STR  R0,C
```

In a sequence where a register is pushed and popped, but no code between the `PUSH` and `POP` uses that *same* register, the `PUSH` and `POP` may be removed:

```
8088 CODE          ARM CODE
MOV  AX,A          LDR  R0,A
MOV  BX,B          LDR  R1,B
ADD  AX,BX         ADD  R0,R0,R1
MOV  C,AX          STR  R0,C
```

The resulting code for the ARM is as small as possible, but the instruction set of the 8088 allows for one further optimization where an operand may be fetched from memory instead of a register:

```
8088 CODE
MOV  AX,A
ADD  AX,B
MOV  C,AX
```

During the compilation process in a traditional compiler, all the modules are called in sequence. As each statement is read in, it is lexically analyzed and parsed, the internal representation is optimized, machine code is generated, peephole optimized, and written to the output stream. Early versions of Turbo Pascal™ skipped the optimization steps in order to speed up compilation; in a primitive "rapid application design" package such as this it was critical to reduce the time between editing a program and running it, particularly since computers were quite slow by today's standards. The efficiency of the executable code was of lesser importance.

Better code can be generated if the entire program is read in before any optimization steps are performed, but this requires significantly more memory and computation time than incremental techniques.

Keeping the entire internal representation of a program in memory allows for some interesting variations. If the proper tools are available to edit the internal representation directly, the lexical analysis and parsing phases can be eliminated. Indeed, the internal representation is the most important part of the entire process, even for

traditional compilers. If different parser "front ends" compile to the same representation, then people can generate compatible programs from different source languages. Similarly, the use of different code generator "back ends" allows the programs to be compiled for different machine architectures from the same internal representation.