A Practical Tool Kit for Making Portable Compilers

Andrew S. Tanenbaum
Hans van Staveren
E. G. Keizer
Johan W. Stevenson

Mathematics Dept.
Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

The Amsterdam Compiler Kit is an integrated collection of programs designed to simplify the task of producing portable (cross) compilers and interpreters. For each language to be compiled, a program (called a front end) must be written to translate the source program into a common intermediate code. This intermediate code can be optimized and then either directly interpreted or translated to the assembly language of the desired target machine. The paper describes the various pieces of the tool kit in some detail, as well as discussing the overall strategy.

Keywords: Compiler, Interpreter, Portability, Translator

CR Categories: 4.12, 4.13, 4.22

Author's present addresses:
A.S. Tanenbaum, H. van Staveren, E.G. Keizer: Mathematics Dept., Vrije Universiteit, Postbus 7161, 1007 MC Amsterdam, The Netherlands

J.W. Stevenson: NV Philips, S&I, T&M, Building TQ V5, Eindhoven, The Netherlands

July 1984

A Practical Tool Kit for Making Portable Compilers

Andrew S. Tanenbaum
Hans van Staveren
E. G. Keizer
Johan W. Stevenson

Mathematics Dept.
Vrije Universiteit
Amsterdam, The Netherlands

*.

Introduction

As more and more organizations acquire many micro- and minicomputers, the need for portable compilers is becoming more and more acute. The present situation, in which each hardware vendor provides its own compilers -- each with its own deficiencies and extensions, and none of them compatible -- leaves much to be desired. The ideal situation would be an integrated system containing a family of (cross) compilers, each compiler accepting a standard source language and producing code for a wide variety of target machines. Furthermore, the compilers should be compatible, so programs written in one language can call procedures written in another language. Finally, the system should be designed so as to make adding new languages and new machines easy. Such an integrated system is being built at the Vrije Universiteit. Its design and implementation is the subject of this article.

Our compiler building system, which is called the "Amsterdam Compiler Kit" (ACK), can be thought of as a "tool kit." It consists of a number of parts that can be combined to form compilers (and interpreters) with various properties. The tool kit is based on an idea (UNCOL) that was first suggested in 1960 [7], but which never really caught on then. The problem which UNCOL attempts to solve is how to make a compiler for each of *N* languages on *M* different machines without having to write *N* x *M* programs.

As shown in Fig. 1, the UNCOL approach is to write *N* "front ends," each of which translates one source language to a common intermediate language, UNCOL (UNiversal Computer Oriented Language), and *M* "back ends," each of which translates programs in UNCOL to a specific machine language. Under these conditions, only *N* + *M* programs must be written to provide all *N* languages on all *M* machines, instead of *N* x *M* programs.

Various researchers have attempted to design a suitable UNCOL [2,8], but none of these have become popular. It is our belief that previous attempts have failed because they have been too ambitious, that is, they have tried to cover all languages and all machines using a single UNCOL. Our approach is more modest: we cater only to algebraic languages and machines whose memory consists of 8-bit bytes, each with its own address. Typical languages that could be handled include Ada, ALGOL 60, ALGOL 68, BASIC, C, FORTRAN, Modula, Pascal, PL/I, PL/M, PLAIN, and RATFOR, whereas COBOL, LISP, and SNOBOL would be less efficient. Examples of machines that could be included are the Intel 8080 and 8086, Motorola 6800, 6809, and 68000, Zilog Z80 and Z8000, DEC PDP-11 and VAX, and IBM 370 but not the Burroughs 6700, CDC Cyber, or Univac 1108 (because they are not byte-oriented). With these restrictions, we believe the old UNCOL idea can be used as the basis of a practical compiler-building system. *.

An Overview of the Amsterdam Compiler Kit

The tool kit consists of eight components:

Fig. 1. The UNCOL model.

1. The preprocessor.
2. The front ends.
3. The peephole optimizer.
4. The global optimizer.
5. The back end.
6. The target machine optimizer.
7. The universal assembler/linker.
8. The utility package.

A fully optimizing compiler, depicted in Fig. 2, has seven cascaded phases. Conceptually, each component reads an input file and writes a transformed output file to be used as input to the next component. In practice, some components may use temporary files to allow multiple passes over the input or internal intermediate files.

Fig. 2. Structure of the Amsterdam Compiler Kit.

In the following paragraphs we will briefly describe each component. After this overview, we will look at all of them again in more detail. A program to be compiled is first fed into the (language independent) preprocessor, which provides a simple macro facility, and similar textual facilties. The preprocessor's output is a legal program in one of the programming languages supported, whereas the input is a program possibly augmented with macros, etc.

This output goes into the appropriate front end, whose job it is to produce intermediate code. This intermediate code (our UNCOL) is the machine language for a simple stack machine called EM (Encoding Machine). A typical front end might build a parse tree from the input, and then use the parse tree to generate EM code, which is similar to reverse Polish. In order to perform this work, the front end has to maintain tables of declared variables, labels, etc., determine where to place the data structures in memory, and so on.

The EM code generated by the front end is fed into the peephole optimizer, which scans it with a window of a few instructions, replacing certain inefficient code sequences by better ones. Such a search is important because EM contains instructions to handle numerous important special cases efficiently (e.g., incrementing a variable by 1). It is our strategy to relieve the front ends of the burden of hunting for special cases because there are many front ends and only one peephole optimizer. By handling the special cases in the peephole optimizer, the front ends become simpler, easier to write and easier to maintain.

Following the peephole optimizer is a global optimizer [5], which unlike the peephole optimizer, examines the program as a whole. It builds a data flow graph to make possible a variety of global optimizations, among them, moving invariant code out of loops, avoiding redundant computations, live/dead analysis and eliminating tail recursion. Note that the output of the global optimizer is still EM code.

Next comes the back end, which differs from the front ends in a fundamental way. Each front end is a separate program, whereas the back end is a single program that is driven by a machine dependent driving table. The driving table for a specific machine tells how the EM code is mapped onto the machine's assembly language. Although a simple driving table might just macro expand each EM instruction into a sequence of target machine instructions, a much more sophisticated translation strategy is normally used, as described later. For speed, the back end does not actually read in the driving table at run time. Instead, the tables are compiled along with the back end in advance, resulting in one binary program per machine.

The output of the back end is a program in the assembly language of some particular machine. The next component in the pipeline reads this program and performs peephole optimization on it. The optimizations performed here involve idiosyncracies of the target machine that cannot be performed in the machine-independent EM-to-EM peephole optimizer. Typically these optimizations take advantage of special instructions or special addressing modes.

The optimized target machine assembly code then goes into the final component in the pipeline, the universal assembler/linker. This program assembles the input to object format, extracting routines from libraries and including them as needed.

The final component of the tool kit is the utility package, which contains various test programs, interpreters for EM code, EM libraries, conversion programs, and other aids for the implementer and user. *.

The Preprocessor

The function of the preprocessor is to extend all the programming languages by adding certain generally useful facilities to them in a uniform way. One of these is a simple macro system, in which the user can give names to character strings. The names can be used in the program, with the knowledge that they will be macro expanded prior to being input to the front end. Macros can be used for named constants, expanding short "procedures" in line, etc.

Another useful facility provided by the preprocessor is the ability to include compile-time libraries. On large projects, it is common to have all the declarations and definitions gathered together in a few files that are textually included in the programs by instructing the preprocessor to read them in, thus fooling the front end into thinking that they were part of the source program.

A third feature of the preprocessor is conditional compilation. The input program can be split up into labeled sections. By setting flags, some of the sections can be deleted by the preprocessor, thus allowing a family of slightly different programs to be conveniently stored on a single file. *.

The Front Ends

A front end is a program that converts input in some source language to a program in EM. At present, front ends exist or are in preparation for Pascal, C, and Plain, and are being considered for Ada, ALGOL 68, FORTRAN 77, and Modula 2. Each of the present front ends is independent of all the other ones, although a general-purpose, table-driven front end is conceivable, provided one can devise a way to express the semantics of the source language in the driving tables. The Pascal front end uses a top-down parsing algorithm (recursive descent), whereas the C and Plain front ends are bottom-up.

All front ends, independent of the language being compiled, produce a common intermediate code called EM, which is the assembly language for a simple stack machine. The EM machine is based on a memory architecture containing a stack for local variables, a (static) data area for variables declared in the outermost block and global to the whole program, and a heap for dynamic data structures. In some ways EM resembles P-code [6], but is more general, since it is intended for a wider class of languages than just Pascal.

The EM instruction set has been described elsewhere [9,10,11] so we will only briefly summarize it here. Instructions exist to:

1. Load a variable or constant of some length onto the stack.
2. Store the top item on the stack in memory.
3. Add, subtract, multiply, divide, etc. the top two stack items.
4. Examine the top one or two stack items and branch conditionally.
5. Call procedures and return from them.

Loads and stores come in several variations, corresponding to the most common programming language semantics, for example, constants, simple variables, fields of a record, elements of an array, and so on. Distinctions are also made between variables local to the current block (i.e., stack frame), those in the outermost block (static storage), and those at intermediate lexicographic levels, which are accessed by following the static chain at run time.

All arithmetic instructions have a type (integer, unsigned, real, pointer, or set) and an operand length, which may either be explicit or may be popped from the stack at run time. Monadic branch instructions pop an item from the stack and branch if it is less than zero, less than or equal to zero, etc. Dyadic branch instructions pop two items, compare them, and branch accordingly.

In addition to these basic EM instructions, there is a collection of special purpose instructions (e.g., to increment a local variable), which are typically produced from the simple ones by the peephole optimizer. Although the complete EM instruction set contains nearly 150 instructions, only about 60 of them are really primitive; the rest are simply abbreviations for commonly occurring EM instruction sequences.

Of particular interest is the way object sizes are parametrized. The front ends allow the user to indicate how many bytes an integer, real, etc. should occupy. Given this information, the front ends can allocate memory, determining the placement of variables within the stack frame. Sizes for primitive types are restricted to 8, 16, 32, 64, etc. bits. The front ends are also parametrized by the target machine's word length and address size so they can tell, for example, how many "load" instructions to generate to move a 32-bit integer. In the examples used henceforth, we will assume a 16-bit word size and 16-bit integers.

Since only byte-addressable target machines are permitted, it is nearly always possible to implement any requested sizes on any target machine. For example, the designer of the back end tables for the Z80 should provide code for 8-, 16-, and 32-bit arithmetic. In our view, the Pascal, C, or Plain programmer specifies what lengths are needed, without reference to the target machine, and the back end provides it. This approach greatly enhances portability. While it is true that doing all arithmetic using 32-bit integers on the Z80 will not be terribly fast, we feel that if that is what the programmer needs, it should be possible to implement it.

Like all assembly languages, EM has not only machine instructions, but also pseudoinstructions. These are used to indicate the start and end of each procedure, allocate and initialize storage for data, and similar functions. One particularly important pseudoinstruction is the one that is used to transmit information to the back end for optimization purposes. It can be used to suggest variables that are good candidates to assign to registers, delimit the scope of loops, indicate that certain variables contain a useful value (next operation is a load) or not (next operation is a store), and various other things. *.

The Peephole Optimizer

The peephole optimizer reads in unoptimized EM programs and writes out optimized ones. Both the input and output are expressed in a highly compact code, rather than in ASCII, to reduce the i/o time, which would otherwise dominate the CPU time. The program itself is table driven, and is, by and large, ignorant of the semantics of EM. The knowledge of EM is contained in a language- and machine-independent table consisting of about 400 pattern-replacement pairs. We will briefly describe the kinds of optimizations it performs below; a more complete discussion can be found in [9].

Each line in the driving table describes one optimization, consisting of a pattern part and a replacement part. The pattern part is a series of one or more EM instructions and a boolean expression. The replacement part is a series of EM instructions with operands. A typical optimization might be:

```
LOL  LOC  ADI  STL  ($1 = $4) and ($2 = 1) and ($3 = 2) ==> INL $1
```

where the text prior to the ==> symbol is the pattern and the text after it is the replacement. LOL loads a local variable onto the stack, LOC loads a constant onto the stack, ADI is integer addition, and STL is store local. The pattern specifies that four consecutive EM instructions are present, with the indicated opcodes, and that furthermore the operand of the first instruction (denoted by $1) and the fourth instruction (denoted by $4) are the same, the constant pushed by LOC is 1, and the size of the integers added by ADI is 2 bytes. (EM instructions have at most one operand, so it is not necessary to specify the operand number.) Under these conditions, the four instructions can be replaced by a single INL (increment local) instruction whose operand is equal to that of LOL.

Although the optimizations cover a wide range, the main ones can be roughly divided into the following categories. *Constant folding* is used to evaluate constant expressions, such as 2*3 + 7 at compile time instead of run time. *Strength reduction* is used to replace one operation, such as multiply, by another, such as shift. *Reordering of expressions* helps in cases like -K/5, which can be better evaluated as K/-5, because the former requires a division and a negation, whereas the latter requires only a division. *Null instructions* include resetting the stack pointer after a call with 0 parameters, offsetting zero bytes to access the first element of a record, or jumping to the next instruction. *Special instructions* are those like INL, which deal with common special cases such as adding one to a variable or comparing something to zero. *Group moves* are useful because a sequence of consecutive moves can often be replaced with EM code that allows the back end to generate a loop instead of in line code. *Dead code elimination* is a technique for removing unreachable statements, possibly made unreachable by previous optimizations. *Branch chain compression* can be applied when a branch instruction jumps to another branch instruction. The first branch can jump directly to the final destination instead of indirectly.

The last two optimizations logically belong in the global optimizer but are in the local optimizer for historical reasons (meaning that the local optimizer has been the only optimizer for many years and the optimizations were easy to do there). *.

The Global Optimizer

In contrast to the peephole optimizer, which examines the EM code a few lines at a time through a small window, the global optimizer examines the program's large scale structure. Three distinct types of optimizations can be found here:

1. Interprocedural optimizations.
2. Intraprocedural optimizations.
3. Basic block optimizations.

We will now look at each of these in turn.

Interprocedural optimizations are those spanning procedure boundaries. The most important one is deciding to expand procedures in line, especially short procedures that occur in loops and pass several parameters. If it takes more time or memory to pass the parameters than to do the work, the program can be improved by eliminating the procedure. The inverse optimization -- discovering long common code sequences and turning them into a procedure -- is also possible, but much more difficult. Like much of the global optimizer's work, the decision to make or not make a certain program transformation is a heuristic one, based on knowledge of how the back end works, how most target machines are organized, etc.

The heart of the global optimizer is its analysis of individual procedures. To perform this analysis, the optimizer must locate the basic blocks, instruction sequences which can be entered only at the top and exited only at the bottom. It then constructs a data flow graph, with the basic blocks as nodes and jumps between blocks as arcs.

From the data flow graph, many important properties of the program can be discovered and exploited. Chief among these is the presence of loops, indicated by cycles in the graph. One important optimization is looking for code that can be moved outside the loop, either prior to it or subsequent to it. Such code motion saves execution time, although it does not save memory. Unrolling loops is also possible and desirable in some cases.

Another area in which global analysis of loops is especially important is in register allocation. While it is true that EM does not have any registers to allocate, the optimizer can easily collect information to allow the back end to allocate registers wisely. For example, the global optimizer can collect static frequency-of-use and live/dead information about variables. (A variable is dead at some point in the program if its current value is not needed, i.e., the next reference to it overwrites it rather than reading it; if the current value will eventually be used, the variable is live.) If two variables are never simultaneously live over some interval of code (e.g., the body of a loop), they can be packed into a single variable, which, if used often enough, may warrant being assigned to a register.

Many loops involve arrays: this leads to other optimizations. If an array is accessed sequentially, with each iteration using the next higher numbered element, code improvement is often possible. Typically, a pointer to the bottom element of each array can be set up prior to the loop. Within the loop the element is accessed indirectly via the pointer, which is also incremented by the element size on each iteration. If the target machine has an autoincrement addressing mode and the pointer is assigned to a register, an array access can often be done in a single instruction.

Other intraprocedural optimizations include removing tail recursion (last statement is a recursive call to the procedure itself), topologically sorting the basic blocks to minimize the number of branch instructions, and common subexpression recognition.

The third general class of optimizations done by the global optimizer is improving the structure of a basic block. For the most part these involve transforming arithmetic or boolean expressions into forms that are likely to result in better target code. As a simple example, A + B*C can be converted to B*C + A. The latter can often be handled by loading B into a register, multiplying the register by C, and then adding in A, whereas the former may involve first putting A into a temporary, depending on the details of the code generation table. Another example of this kind of basic block optimization is transforming -B + A < 0 into the equivalent, but simpler, A < B. *.

The Back End

The back end reads a stream of EM instructions and generates assembly code for the target machine. Although the algorithm itself is machine independent, for each target machine a machine dependent driving table must be supplied. The driving table effectively defines the mapping of EM code to target code.

It will be convenient to think of the EM instructions being read as a stream of tokens. For didactic purposes, we will concentrate on two kinds of tokens: those that load something onto the stack, and those that perform some operation on the top one or two values on the stack. The back end maintains at compile time a simulated stack whose behavior mirrors what the stack of a hardware EM machine would do at run time. If the current input token is a load instruction, a new entry is pushed onto the simulated stack.

Consider, as an example, the EM code produced for the statement K := I + 7. If K and I are 2-byte local variables, it will normally be LOL I; LOC 7; ADI 2; STL K. Initially the simulated stack is empty. After the first token has been read and processed, the simulated stack will contain a stack token of type MEM with attributes telling that it is a local, giving its address, etc. After the second token has been read and processed, the top two tokens on the simulated stack will be CON (constant) on top and MEM directly underneath it.

At this point the back end reads the ADI 2 token and looks in the driving table to find a line or lines that define the action to be taken for ADI 2. For a typical multiregister machine, instructions will exist to add constants to registers, but not to memory. Consequently, the driving table will not contain an entry for ADI 2 with stack configuration CON, MEM.

The back end is now faced with the problem of how to get from its current stack configuration, CON, MEM, which is not listed, to one that is listed. The table will normally contain rules (which we call "coercions") for converting between CON, REG, MEM, and similar tokens. Therefore the back end attempts to "coerce" the stack into a configuration that *is* present in the table. A typical coercion rule might tell how to convert a MEM into a REG, namely by performing the actions of allocating a register and emitting code to move the memory word to that register. Having transformed the compile-time stack into a configuration allowed for ADI 2, the rule can be carried out. A typical rule for ADI 2 might have stack configuration REG, MEM and would emit code to add the MEM to the REG, leaving the stack with a single REG token instead of the REG and MEM tokens present before the ADI 2.

In general, there will be more than one possible coercion path. Assuming reasonable coercion rules for our example, we might be able to convert CON MEM into CON REG by loading the variable I into a register. Alternatively, we could coerce CON to REG by loading the constant into a register. The first coercion path does the add by first loading I into a register and then adding 7 to it. The second path first loads 7 into a register and then adds I to it. On machines with a fast LOAD IMMEDIATE instruction for small constants but no fast ADD IMMEDIATE, or vice versa, one code sequence will be preferable to the other.

In fact, we actually have more choices than suggested above. In both coercion paths a register must be allocated. On many machines, not every register can be used in every operation, so the choice may be important. On some machines, for example, the operand of a multiply must be in an odd register. To summarize, from any state (i.e., token and stack configuration), a variety of choices can be made, leading to a variety of different target code sequences.

To decide which of the various code sequences to emit, the back end must have some information about the time and memory cost of each one. To provide this information, each rule in the driving table, including coercions, specifies both the time and memory cost of the code emitted when the rule is applied. The back end can then simply try each of the legal possibilities (including all the possible register allocations) to find the cheapest one.

This situation is similar to that found in a chess or other game-playing program, in which from any state a finite number of moves can be made. Just as in a chess program, the back end can look at all the "moves" that can be made from each state reachable from the original state, and thus find the sequence that gives the minimum cost to a depth of one. More generally, the back end can evaluate all paths corresponding to accepting the next *N* input tokens, find the cheapest one, and then make the first move along that path, precisely the way a chess program would.

Since the back end is analogous to both a parser and a chess playing program, some clarifying remarks may be helpful. First, chess programs and the back end must do some look ahead, whereas the parser for a well-designed grammar can usually suffice with one input token because grammars are supposed to be unambiguous. In contrast, many legal mappings from a sequence of EM instructions to target code may exist. Second, like a parser but unlike a chess program, the back end has perfect information -- it does not have to contend with an unpredictable opponent's moves. Third, chess programs normally make a static evaluation of the board and label the *nodes* of the tree with the resulting scores. The back end, in contrast, associates costs with *arcs* (moves) rather than nodes (states). However, the difference is not essential, since it could also label each node with the cumulative cost from the root to that node.

As mentioned above, the cost field in the table contains *both* the time and memory costs for the code emitted. It should be clear that the back end could use either one or some linear combination of them as the scoring function for evaluating moves. A user can instruct the compiler to optimize for time or for memory or for, say, 0.3 x time + 0.7 x memory. Thus the same compiler can provide a wide range of performance options to the user. The writer of the back end table can take advantage of this flexibility by providing several code sequences with different tradeoffs for each EM instruction (e.g., in line code vs. call to a run time routine).

In addition to the time-space tradeoffs, by specifying the depth of search parameter, *N*, the user can effectively also tradeoff compile time vs. object code quality, for whatever code metric has been chosen. In summary, by combining the properties of a parser and a game playing program, it is possible to make a code generator that is table driven, highly flexible, and has the ability to produce good code from a stack machine intermediate code. *.

The Target Machine Optimizer

In the model of Fig 2., the peephole optimizer comes before the global optimizer. It may happen that the code produced by the global optimizer can also be improved by another round of peephole optimization. Conceivably, the system could have been designed to iterate peephole and global optimizations until no more of either could be performed.

However, both of these optimizations are done on the machine independent EM code. Neither is able to take advantage of the peculiarities and idiosyncracies with which most target machines are well endowed. It is the function of the final optimizer to do any (peephole) optimizations that still remain.

The algorithm used here is the same as in the EM peephole optimizer. In fact, if it were not for the differences between EM syntax, which is very restricted, and target assembly language syntax, which is less so, precisely the same program could be used for both. Nevertheless, the same ideas apply concerning patterns and replacements, so our discussion of this optimizer will be restricted to one example.

To see what the target optimizer might do, consider the PDP-11 instruction sequence sub #2,r0; mov (r0),x. First 2 is subtracted from register 0, then the word pointed to by it is moved to x. The PDP-11 happens to have an addressing mode to perform this sequence in one instruction: mov -(r0),x. Although it is conceivable that this instruction could be included in the back end driving table for the PDP-11, it is awkward to do so because it can occur in so many contexts. It is much easier to catch things like this in a separate program. *.

The Universal Assembler/Linker

Although assembly languages for different machines may appear very different at first glance, they have a surprisingly large intersection. We have been able to construct an assembler/linker that is almost entirely independent of the assembly language being processed. To tailor the program to a specific assembly language, it is necessary to supply a table giving the list of instructions, the bit patterns required for each one, and the language syntax. The machine independent part of the assembler/linker is then compiled with the table to produce an assembler and linker for a particular target machine. Experience has shown that writing the necessary table for a new machine can be done in less than a week.

To enforce a modicum of uniformity, we have chosen to use a common set of pseudoinstructions for all target machines. They are used to initialize memory, allocate uninitialized memory, determine the current segment, and similar functions found in most assemblers.

The assembler is also a linker. After assembling a program, it checks to see if there are any unsatisfied external references. If so, it begins reading the libraries to find the necessary routines, including them in the object file as it finds them. This approach requires libraries to be maintained in assembly language form, but eliminates the need for inventing a language to express relocatable object programs in a machine independent way. It also simplifies the assembler, since producing absolute object code is easier than producing relocatable object code. Finally, although assembly language libraries may be somewhat larger than relocatable object module libraries, the loss in speed due to having more input may be more than compensated for by not having to pass an intermediate file between the assembler and linker. *.

The Utility Package

The utility package is a collection of programs designed to aid the implementers of new front ends or new back ends. The most useful ones are the test programs. For example, one test set, EMTEST, systematically checks out a back end by executing an ever larger subset of the EM instructions. It starts out by testing LOC, LOL and a few of the other essential instructions. If these appear to work, it then tries out new instructions one at a time, adding them to the set of instructions "known" to work as they pass the tests.

Each instruction is tested with a variety of operands chosen from values where problems can be expected. For example, on target machines which have 16-bit index registers but only allow 8-bit displacements, a fundamentally different algorithm may be needed for accessing the first few bytes of local variables and those with offsets of thousands. The test programs have been carefully designed to thoroughly test all relevant cases.

In addition to EMTEST, test programs in Pascal, C, and other languages are also available. A typical test is:

i := 9; **if** i + 250 <> 259 **then** error(16);

Like EMTEST, the other test programs systematically exercise all features of the language being tested, and do so in a way that makes it possible to pinpoint errors precisely. While it has been said that testing can only demonstrate the presence of errors and not their absence, our experience is that the test programs have been invaluable in debugging new parts of the system quickly.

Other utilities include programs to convert the highly compact EM code produced by front ends to ASCII and vice versa, programs to build various internal tables from human writable input formats, a variety of libraries written in or compiled to EM to make them portable, an EM assembler, and EM interpreters for various machines.

Interpreting the EM code instead of translating it to target machine language is useful for several reasons. First, the interpreters provide extensive run time diagnostics including an option to list the original source program (in Pascal, C, etc.) with the execution frequency or execution time for each source line printed in the left margin. Second, since an EM program is typically about one-third the size of a compiled program, large programs can be executed on small machines. Third, running the EM code directly makes it easier to pinpoint errors in the EM output of front ends still being debugged. *.

Summary and Conclusions

The Amsterdam Compiler Kit is a tool kit for building portable (cross) compilers and interpreters. The main pieces of the kit are the front ends, which convert source programs to EM code, optimizers, which improve the EM code, and back ends, which convert the EM code to target assembly language. The kit is highly modular, so writing one front end (and its associated runtime routines) is sufficient to implement a new language on a dozen or more machines, and writing one back end table and one universal assembler/linker table is all that is needed to bring up all the previously implemented languages on a new machine. In this manner, the contents, and hopefully the usefulness, of the toolkit will increase in time.

We believe the principal lesson to be learned from our work is that the old UNCOL idea is basically a sound way to produce compilers, provided suitable restrictions are placed on the source languages and target machines. We also believe that although compilers produced by this technology may not be equal to the very best handcrafted compilers, in terms of object code quality, they are certainly competitive with many existing compilers. However, when one factors in the cost of producing the compiler, the possible slight loss in performance may be more than compensated for by the large decrease in production cost. As a consequence of our work and similar work by other researchers [1,3,4], we expect integrated compiler building kits to become increasingly popular in the near future.

The toolkit is now available for various computers running the UNIX® operating system. For information, contact the authors. *.

References

1. Graham, S.L. Table-Driven Code Generation. *Computer 13*, 8 (August 1980), 25-34.

A discussion of systematic ways to do code generation, in particular, the idea of having a table with templates that match parts of the parse tree and convert them into machine instructions.

2. Haddon, B.K., and Waite, W.M. Experience with the Universal Intermediate Language Janus. *Software Practice & Experience 8*, 5 (Sept.-Oct. 1978), 601-616.

An intermediate language for use with ALGOL 68, Pascal, etc. is described. The paper discusses some problems encountered and how they were dealt with.

3. Johnson, S.C. A Portable Compiler: Theory and Practice. *Ann. ACM Symp. Prin. Prog. Lang.*, Jan. 1978.

A cogent discussion of the portable C compiler. Particularly interesting are the author's thoughts on the value of computer science theory.

4. Leverett, B.W., Cattell, R.G.G, Hobbs, S.O., Newcomer, J.M., Reiner, A.H., Schatz, B.R., and Wulf, W.A. An Overview of the Production-Quality Compiler-Compiler Project. *Computer 13*, 8 (August 1980), 38-49.

PQCC is a system for building compilers similar in concept but differing in details from the Amsterdam Compiler Kit. The paper describes the intermediate representation used and the code generation strategy.

5. Lowry, E.S., and Medlock, C.W. Object Code Optimization. *Commun. ACM 12*, (Jan. 1969), 13-22.

A classic paper on global object code optimization. It covers data flow analysis, common subexpressions, code motion, register allocation and other techniques.

6. Nori, K.V., Ammann, U., Jensen, K., Nageli, H. The Pascal P Compiler Implementation Notes. Eidgen. Tech. Hochschule, Zurich, 1975.

A description of the original P-code machine, used to transport the Pascal-P compiler to new computers.

7. Steel, T.B., Jr. UNCOL: the Myth and the Fact. in *Ann. Rev. Auto. Prog.* Goodman, R. (ed.), vol 2., (1960), 325-344.

An introduction to the UNCOL idea by its originator.

8. Steel, T.B., Jr. A First Version of UNCOL. *Proc. Western Joint Comp. Conf.*, (1961), 371-377.

The first detailed proposal for an UNCOL. By current standards it is a primitive language, but it is interesting for its historical perspective.

9. Tanenbaum, A.S., van Staveren, H., and Stevenson, J.W. Using Peephole Optimization on Intermediate Code. *ACM Trans. Prog. Lang. and Sys. 3*, 1 (Jan. 1982) pp. 21-36.

A detailed description of a table-driven peephole optimizer. The driving table provides a list of patterns to match as well as the replacement text to use for each successful match.

10. Tanenbaum, A.S., Stevenson, J.W., Keizer, E.G., and van Staveren, H. Description of an Experimental Machine Architecture for use with Block Structured Languages. Informatica Rapport 81, Vrije Universiteit, Amsterdam, 1983.

The defining document for EM.

11. Tanenbaum, A.S. Implications of Structured Programming for Machine Architecture. *Comm. ACM 21*, 3 (March 1978), 237-246.

The background and motivation for the design of EM. This early version emphasized the idea of interpreting the intermediate code (then called EM-1) rather than compiling it.