Machine code generation within a compiler—a teaching model

L. V. Atkinson* and G. J. Duffus†

A model is presented wherein aspects of the code generation phase of a compiler can be taught practically. SLPL (Atkinson, 1976) is a high level language with the facility to generate and execute sequences of machine code orders on ICL 1900 series machines. This paper describes the implementation of this facility and illustrates its application.

(Received May 1975)

The different actions of a typical two-pass compiler for an ALGOL type language can be summarised as follows.

First pass:
(a) lexical compaction
(b) syntax analysis
(c) construction of semantic tables
(d) transformation to intermediate form.

Second pass:
(e) semantic checking
(f) machine code generation.

As indicated some of these activities proceed in parallel but each may be more easily understood if treated independently. It is suggested that an undergraduate course in compiling techniques might treat each aspect separately before discussing ways in which they can be combined. We examine here in detail how the code generation phase can be implemented independently and outline other aspects only where relevant.

Associated with any compiler are at least four language levels. A multi-pass compiler may use several different intermediate forms and so have more than four.

1. Implementation language (the language in which the compiler is written).

In the present case this is SLPL. The system to be described is an integral part of SLPL but a similar scheme could be incorporated within any language by supplying a few code-bodied procedures. There are three features of SLPL not present in all languages which make its use particularly suitable.

1. Recursion

Recursion is not necessary for compiler writing but its use makes the techniques easier to understand and to apply. Without recursion one tends not to see the wood for the trees!

2. String variables

The availability of string variables may not seem at first to be of any great importance but their use allows a comprehensible representation of the input data for the code generator and simplifies the input module when such a representation is used. A source program contains identifiers (strings) and basic symbols (strings) but during the first pass of the compiler their representation is changed to that of the intermediate code. Each basic symbol, regardless of length, is reduced to a single unique (short) bit pattern and each identifier is replaced by a pointer to an appropriate part of the semantic tables. With such an intermediate code form it is not easy to treat the code generator independently because of the difficulty of supplying appropriate data. By retaining all such strings in their source form the intermediate code becomes simply a slight modification of the source language.

A further advantage of string variables presents itself when the generation of individual machine code orders is considered. A machine code function is usually referred to by a mnemonic opcode, a character string. Using string variables we can denote the function part of an order by its opcode.

3. List structures

To produce code for an arithmetic or boolean expression we need to know its structure. This is normally implicitly represented in intermediate code by some form of Polish Representation (Randall and Russell, 1964). Again independent treatment of the code generator is hampered because of the unnatural form of the intermediate code. Further, the generation of machine code by an algorithm working from a Polish form expression is not, in the authors’ opinion, so natural as by an algorithm which can make use of an explicit representation of structure. This point is illustrated by the example program listed at the end of this paper.

Explicit structure representation within the intermediate code is not difficult to produce when one comes to implement the syntax analyser. The analyser can be top-down and recursive without back-up but with expressions treated by operator precedence techniques. Adopting such an approach and using a language which allows the use of structures it may well be easier to produce intermediate code of the form described rather than transforming expressions into Polish notation.

2. Source language (the language to be compiled).

The principles of code generation are best approached in stages. Assignment statements should be considered before conditionals and loops and simple variables before arrays and procedures. We take ALGOL 60 as the source language and accordingly approach compilation by defining subsets of the language, each successive subset containing the previous subset. The key to creating a simple code generation system is the provision of a suitable run-time model, within which the object code is executed, associated with a correspondingly simple source language subset. Such a model is described in the following sections.

3. Intermediate code (output from the first pass, which is also input to the code generator).

In the light of the above discussion of the implementation language we can supply as input to the code generator a program written in a slightly modified form of the source language. Identifiers and basic symbols are to be somehow delimited as strings and the structure of expressions is to be explicitly represented. The input conventions of SLPL are

*Department of Applied Mathematics and Computing Science, The University, Sheffield S10 2TN.
†Department of Computing Services, The University, Sheffield S10 2TN.
particularly suited to this. The input module of an SLPL program will accept atoms or lists. Relevant to the present context are lists (sequences of atoms or lists suitably bracketed and separated—the external representation of the brackets and separator may be specified by the user or the separator may be suppressed) and atoms of mode string (a character string enclosed between ‘ ’ and ‘ ’) and char (any character, apart from the space and end of record characters, which is not a list bracket, a list element separator or part of any other atom (string, int or bool) is taken to be a character atom). A simple input representation now suggests itself. Basic symbols can be strings, identifiers can be single characters and, if we assume all operators are dyadic, expressions can be fully bracketed triplets. It is convenient to treat the assignment operator (denoted, say, $\leftarrow$) as other operators and to bracket it with its operands. If the list element separator is suppressed the following program indicates a possible form of input data with strings represented in bold face.

```
begin
  integer $i, j, k$; boolean $b$;
  read $i$; read $j$;
  print $i$; print $j$;
  ($b \leftarrow (i = j)$);
  if $b$ then read $k$ else ($k \leftarrow (i + j)$);
  print $k$
end
```

Any suitable convention for the form of source and intermediate language I/O statements may be adopted. The above example indicates one of the simplest.

4. Object code (the output from the code generator).

A comprehensive undergraduate course dealing with compiling techniques might well consider the generation of a stack—oriented object code for a hypothetical machine (Randell and Russell, 1963). For a practical implementation it is a simple matter to write an emulator of such a machine. This has the advantage that the overall structure of the code generator can be considered without going into details of production of strict machine code.

The scheme to be described here, however, utilises a standard assembly language PLAN (ICL, 1972) rather than a hypothetical code. With this approach some basic code generation concepts can be implemented very quickly if students are already familiar with the assembly language.

In a teaching environment emphasis must be placed on error detection and system diagnostics. This suggests that even if the object code is the binary code of the machine it might be interpreted rather than obeyed directly. In the present instance the object code produced is, apart from I/O instructions, ICL 1900 machine code and to provide run-time security as well as comprehensive error checks and diagnostics this code is interpreted. The interpreter is described later and other possible schemes to execute the generated code are outlined and their drawbacks are indicated.

The simple model

1. Source language subset

The code generation phase of a compiler embraces three logically distinct processes.

1. Code is produced to reflect the structure of the program. For example branch instructions are generated to achieve the desired flow of control through a conditional statement or a loop.

2. Semantic processing must be performed on the use of each identifier

   (a) to check that its use is appropriate, and

   (b) to determine the action to be carried out. The action required for a simple variable, for example, differs from that needed for a function designator.

3. Addresses of variables must be obtained. For a simple variable this may be a direct address if the variable is declared in the outer block, a displacement from the start address of the current block if the variable is local, or may be in the form of a block number and an associated displacement within that block.

For the simple model

(a) assume all variables are of type integer or boolean and that their use is always correct and hence ignore (2) above,

(b) dispense with declarations, and

(c) assume all identifiers refer to distinct variables which may be accessed directly. If identifiers must be single letters this implies that 26 variables are available.

In the first instance a small subset of ALGOL statements should be considered and it is suggested that initially the subset contains only compound statements containing simple I/O instructions and assignments statements and that operators are restricted to + and -. It is then a simple step to introduce conditional statements and loops and to extend the set of operators to include some boolean and relational operators.

2. Run-time model

The run-time model provides instruction and data areas and special registers for the object program and an interface between the high level and the low level facets of the interpretive execution phase.

The high level aspects involve

(a) the instruction area, the number of words allocated being specified by the user,

(b) an executive control loop for the interpretation,

(c) I/O facilitating transput of integers.

The low level aspects concern the interpreting routines activated by the executive control loop and the data areas and special registers they may affect.

The interface comprises five words:

(a) CIP the current instruction pointer indicating the next instruction to be obeyed,

(b) INST the instruction currently being obeyed,

(c) LIMIT the address of the last program instruction,

(d) IOVALUE to contain any value currently involved in I/O,

(e) STATUS to indicate error conditions etc.

PLAN utilises eight accumulators and two single bit registers for carry and overflow. Accordingly these are simulated in the model. 26 words are allocated to correspond to the variables $A$ through $Z$. Small arrays can be accommodated within this area.

Expression evaluation necessitates the use of a stack. For the simple model the accumulators can be used provided the position of the stack is known at compile-time (i.e. procedure calls are omitted). For a dynamic stack an index register can be used as the stack pointer and the stack can exist within the words $A$ through $Z$.

3. Extensions to the model

To accommodate larger arrays and a more realistic run-time stack the data space can be extended to say 200 words, the first 26 still being referred to as $A, B, \ldots, Z$. The run-time stack could then start at say $A + 100$ at the user’s discretion. Since all instructions are interpreted there is wide scope for expansion of the instruction set. In particular the I/O facilities can be extended. Further mention of this is made later.
Planting orders

An order is planted by use of the predefined procedure \texttt{plant}. \texttt{plant} has one of the modes

\begin{verbatim}
proc (string)
proc (string, atom)
proc (string, char, int)
proc (string, int, atom)
proc (string, int, atom, int)
\end{verbatim}

and accordingly may be supplied with one, two, three or four actual parameters. The PLAN instruction specified is assembled in the next available instruction word on the code area stack. Adopting the notation \(f(\text{mode}\string)\) for function, \(x(\text{mode}\int)\) for accumulator, \(n(\text{mode}\int\text{or char})\) for operand and \(m(\text{mode}\int)\) for modifier, acceptable formats corresponding to the permissible modes are

\begin{verbatim}
plant(f, x, n) and plant(f, x, n, m)
\end{verbatim}

A call of \texttt{plant} causes the appropriate machine code order to be generated in the word currently accessed by the stack pointer and the stack pointer to be incremented by 1. Initially the code stack pointer is set to zero and may be reset to zero at any time by a call of the system primitive \texttt{reset}.

For forward branches use is made of \texttt{dummy address}, a predefined integer variable:

\begin{verbatim}
e.g. plant (BZE, 4, dummy address)
\end{verbatim}

for only then can the address part of the instruction word generated be overwritten at a later date. To do this its address must be known. \texttt{current address} is an \texttt{int proc} which delivers the address of the location into which the next instruction generated will be planted. Normally a call of \texttt{plant} using \texttt{dummy address} is immediately preceded by an instruction of the form

\begin{verbatim}
fwdjump := current address
\end{verbatim}

and \texttt{fwdjump} then records the address of the forward jump instruction. The address part of this instruction can then be overwritten later using

\begin{verbatim}
proc plant operand (int source, destination)
\end{verbatim}

perhaps by a call of the form

\begin{verbatim}
plant operand (fwdjump, current address)
\end{verbatim}

The operand of a branch instruction must be either \texttt{dummy address} or an address within the allocated code instruction area. The user may specify the size of the code instruction area and assumes a start address of zero. When the code is obeyed a check is made to ensure that the address is within the range of instructions planted. When an attempt is made to overwrite the address part of an instruction word a check is made to ensure that the address part contains the dummy address. A check is also made to ensure that all dummy addresses have been overwritten before the code is obeyed.

3. Input/output

Two opcodes \texttt{RD} and \texttt{PRNT} are supplied to deal with input and output. They are generated using \texttt{plant} and in each case the accumulator field is empty. The effect of

\begin{verbatim}
RD N(M)
\end{verbatim}

is to read the next integer from the input stream and store it in the specified location. The effect of

\begin{verbatim}
PRNT N(M)
\end{verbatim}

is to print out in integer format the contents of the specified location. These instructions are represented in one word by using as the function part of the instruction a bit pattern which the user cannot otherwise generate. In particular there are certain function codes which do not represent legal machine code orders. Use of these illegal order codes in this context allows for future expansion of the I/O functions implemented.

4. Code listing

Each instruction generated is printed in symbolic form as it is planted. This has the disadvantage that forward jumps may be incomplete but has the advantage that whether or not a program runs to completion a complete listing is available of any code planted. A list is also provided of all the updated branch operands.

Output is produced on three separate streams.

\begin{itemize}
  \item \texttt{Stream 1}:
    \begin{itemize}
      \item (a) listing of user program
      \item (b) any compile time errors
    \end{itemize}
\end{itemize}
any run-time error independent of generation or execution of code.

Stream 2:
Copy of data input via read statements in user program.

Stream 3:
(a) a listing of the code program (including dummy addresses) and updated branch operands
(b) any error messages associated with planting code
(c) output from the code program
(d) any execution error generated whilst obeying the code.

The violation of any error check concerned with the planting of code does not usually constitute a fatal error. A warning is given and an illegal function is planted instead of the intended instruction. This is then trapped at run time if an attempt is made to obey the illegal order.

Execution of the code
1. Non-interpretive schemes
Before the interpretive approach to executing the code was implemented two other schemes were used. The first, rather crude method was simply a call to a PLAN segment which branched to the array containing the generated instructions and on completion of the execution control was returned to the user program. This has the obvious disadvantage of total lack of security and as many generated programs had errors in them chaos resulted when parts of the user program were overwritten. A further drawback was that high level I/O could not be so conveniently incorporated.

The second scheme contracted the user program so that only the generated code remained in core and then transferred control to it. This had the advantage that there was no user program to corrupt and no 'spurious' error messages were generated but, clearly, control could not be returned to the user program.

Both methods were very poor on run-time error diagnostics, the only information produced being ILLEGAL messages from the 1900 Executive. Direct execution of the code was therefore replaced by interpretation.

2. Transfer of control
Control is transferred to the code program using

\[
\text{proc enter code (int entry address) .}
\]

This procedure calls the interpreter for the machine code and will commence interpretation with the instruction stored at the address specified. If an error occurs when the code is being obeyed suitable diagnostic information is produced. The error message is accompanied by a printout of the offending instruction, the contents of the accumulators and data area, and a recent trace history of branches. In the event of an error interpretation of the code program is abandoned.

Before interpretation commences the contents of the accumulators and data area are marked as undefined. During interpretation any reference to an undefined value constitutes an error condition. Upon completion of the code program whether naturally or after an error control is returned to the user program.

3. The interpreter
The interpreter is in the form of a set of PLAN segments which are repeatedly called from an executive loop written in ALGOL 60 until the interpreted program terminates normally or an error condition arises. The status word is set by the interpreter to indicate the executive loop the manner of any error or the natural end of the program.

Instructions are stored in an ALGOL array. The executive loop fetches the current instruction to be interpreted and increments the instruction pointer by 1. If this is an I/O instruction it is dealt with at the ALGOL level as described later; if not then the interpreter is called. For instructions which are not branches the interpreter checks that addresses specified are within the program's data area, checks that any values referred to are not undefined, restores the program's accumulators and special registers and, using the PLAN instruction OBEY*, executes the instruction. The accumulators and special registers are then stored. Instructions which would suspend or delete the program being interpreted are not executed but the status word is set accordingly.

Branch instructions are dealt with rather differently since, if a branch is OBEYed, control will not return to the instruction immediately following the OBEY. This means that the interpreter would lose control and chaos would probably result. Instead (see Fig. 1) the address part of the instruction being interpreted (INST) is overwritten with the address (α) of an instruction which resets the current instruction pointer to the address originally specified in INST. The altered branch instruction is then OBEYed. If the branch is successful control is transferred to α. If the branch is an unsuccessful conditional branch control returns to the instruction immediately following the OBEY. Total security is assured with this method.

As mentioned earlier I/O instructions are recognised in the executive loop. Equivalent 1900 ALGOL I/O procedures are used to read and print data values in conjunction with two PLAN segments which transfer the datum between the specified location and I0VALUE. I/O is implemented at a high level because at the machine code level it can be tedious and is not really the concern of the compiler writer.

The system currently implemented embraces all the extensions to the model mentioned and has proven to be quite satisfactory.

Application
To illustrate the ease of writing and testing a code generator the syntax of a simple intermediate code for a subset of ALGOL 60 is specified and a code generator written directly from this. It is assumed that the intermediate code is semantically correct. This code would normally be produced by the compiler and so should be syntactically correct. In this case the code is supplied by the user and so a few simple syntactic checks are made.

*OBEY obeys a specified instruction. If the instruction obeyed is a successful branch then control is transferred to the branch address. If a normal order or unsuccessful branch is obeyed control returns to the instruction following the OBEY.
Intermediate code

The syntax of the program input to the code generator is defined by the following BNF productions.

\[
\begin{align*}
\text{<program>} & ::= \text{<compound st>} \\
\text{<compound st>} & ::= \text{begin <st list> end} \\
\text{<st list>} & ::= \text{<st list><statement>} | \text{<statement>} \\
\text{<statement>} & ::= \text{<compound st>} | \text{<conditional st>} | \text{<i/o>} | \text{<assignment>} \\
\text{<conditional st>} & ::= \text{if <exp> then <statement> else <statement>} \\
\text{<i/o>} & ::= \text{<read <id> | print <exp>} \\
\text{<assignment>} & ::= \text{<identifier> := <exp>} \\
\text{<exp>} & ::= <\text{<exp> <operator> <exp>}> | \text{<identifier>} | \text{<digit>} | \text{<bool>} \\
\text{<identifier>} & ::= A | B | C | D \ldots | Z \\
\text{<operator>} & ::= + | - | and | or | \neq \\
\text{<digit>} & ::= 0 | 1 | 2 \ldots | 9 \\
\text{<bool>} & ::= \text{true} | \text{false}
\end{align*}
\]

Representation

1. true and false are standard SLPL data objects and so have a prescribed representation.

2. All other bold face symbols are represented as alphabetic strings enclosed between " and ".

3. All other terminal symbols are characters represented as in the productions.

4. (" and ") are to be treated as list brackets so that <assignment>s and <exp>s including an <operator> will be stored as list structures.

Example program

Expression evaluation is carried out using the accumulators as the operand stack.

Aspects of SLPL not previously mentioned and not obvious from the context are explained by comments (enclosed between " and ").

The productions, upper case letters are used to represent string constants e.

\[
\text{<symbol> ::= true | false}
\]

\[
\text{<assignment> ::= <identifier> := <exp>}
\]

\[
\text{<exp> ::= <exp> <operator> <exp> | <identifier> | <digit> | <bool>}
\]

\[
\text{<identifier> ::= A | B | C | D \ldots | Z}
\]

\[
\text{<operator> ::= + | - | and | or | \neq}
\]

\[
\text{<digit> ::= 0 | 1 | 2 \ldots | 9}
\]

\[
\text{<bool> ::= true | false}
\]

\[
\text{<procedure> ::= \text{begin}}
\]

\[
\text{<statement> ::= \text{<compound st> | <conditional st> | <i/o> | <assignment>}}
\]

\[
\text{<conditional st> ::= \text{if <exp> then <statement> else <statement>}}
\]

\[
\text{<i/o> ::= \text{<read <id> | print <exp>}}
\]

\[
\text{<assignment> ::= \text{<identifier> := <exp>}}
\]

\[
\text{<exp> ::= <exp> <operator> <exp> | <identifier> | <digit> | <bool>}
\]

\[
\text{<identifier> ::= A | B | C | D \ldots | Z}
\]

\[
\text{<operator> ::= + | - | and | or | \neq}
\]

\[
\text{<digit> ::= 0 | 1 | 2 \ldots | 9}
\]

\[
\text{<bool> ::= true | false}
\]

The Computer Journal
if read = BEGIN then (compound st (read); plant (DEL)) else fail (PROGRAM DOES NOT START WITH 'BEGIN') fi
end c of compile program c;
end.

References


Book review


This book describes itself as a 'Report of a Conference held in Hanover on 28-30 March 1974'. The conference comprised five half day sessions, the first and second sessions being further sub-divided into two parts. Each session or part-session is preceded, in the publication, by a summary, written after the conference, of the papers presented in that session and a synopsis of the discussions which took place. This is a method quite different from that used elsewhere, wherein the discussion is usually presented at the end of each session and then usually it takes the form of speakers being identified together with their questions, answers, or comments.

The 'Table of Contents' lists the papers in presentation order, within the five half-day subdivisions. This is then followed by the list of authors in alphabetical order with their institutions. From this one sees that the only speakers from outside Germany came from Vienna (two), Milan (one) and Chicago (one).

In his summary of Part I of the first half day, which is devoted to 'Information and Communication', Professor Koeppe says that the papers of the first morning were 'characterised by a certain thoughtfulness, a pause for thinking, after perhaps many years of work with too great a technological orientation'. This phraseology succinctly covers the papers in that section as they are both devoted to abstract ideas in communication and information flow, the second being in the area of project management. The second Part of the first half day continues the theme and deals with the use of cybernetics in (a) MANWARE for hospital information systems; (b) the development of social-psychological criteria for the introduction of EDP into hospital procedures; (c) the Hanover Medical System and (d) laboratory data processing, again at the Hanover Medical High School. The only quantitative information occurs in four consecutive pages and is concerned with data storage in the Hanover Medical System. The programming for the system was done in PL/I and the first patient was processed in July 1971.

In the second half-day, practical applications were considered. These included (a) the use of an LP model in a Diagnostic Clinic—found to be severely limited because of an inability to solve the mixed-integer type of LP problem; (b) a critical analysis of the advantages of the use of bar-code in a hospital situation (they are still not convinced of any advantage and claim further work needs to be done; this contrasts with our own situation where bar-code readers are successfully and advantageously used at the North Staffordshire General Hospital at Stoke); (c) monitoring and control of a laboratory system (this paper has no references attached and seems to be describing their progress: it would appear from the description that they are following the lines of the work done at the Queen Elizabeth Hospital, Birmingham, and University College, some ten years ago); (d) identification of patient specimens in laboratories. (The technique is to use magnetised plastic rings which encircle the specimen tubes, the rings carrying 18 bits of information and being read automatically). The next half day was perhaps the most interesting from a practical point of view as it dealt with patient monitoring and patient measurement. The first paper reminds us of the principle of electrode potential and gives a reference list of values for 19 elements. The third paper is concerned with handling false alarm situations in an intensive care unit. Great emphasis is placed on detecting exactly where the signals can become disrupted. The author measures heart-rate, respiratory-rate, and temperature but does not attempt to use the 'two-out-of-three' system such as was developed by Dr. Stewart at Wigan several years earlier and which is now commercially available. They do however store patient data for the previous 24 hours for retrieval on varying time axes, as per the Thoracic Clinic at the Karolinska. The paper which follows describes an attempt to build a servo system for regulation of drip infusion by monitoring urine output. They conclude that a digital computer is essential as the analogue will not cope. The last paper in that session deals with the use of the VDU (even in colour) as a device for display of tomographic type information. It points out that the colour can be distracting in certain cases and resorts to the grey scale where necessary.

The remaining two sessions were devoted to (a) quality control in the biochemistry laboratory, and (b) text processing. Only eleven of the twenty-six papers presented give lists of references. Of these, only five contain any reference to works published in English, and whilst mention is made of various English speaking conferences, sadly there is no mention of any of the many held in the UK. This book, which contains several typographical errors, shows how the Germans are progressing slowly but with assured confidence at each step. They would obviously have learnt a good deal from the papers presented at MEDINFO 74 which followed their conference. The book does credit to the two editors, Professor Peter Reichertz and Dr. Gabriele Holthoff both from the Medical High School at Hanover.

B. Richards (Manchester)