Information Technology Division

RESEARCH REPORT
ERL-0571-RR

DECOMPILING WITH DEFINITE CLAUSE GRAMMARS

by

S. T. Hood

ELECTRONICS RESEARCH LABORATORY

AR-006-765

DSTO AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

RESEARCH REPORT
ERL-0571-RR

DECOMPILING WITH DEFINITE CLAUSE GRAMMARS

by

S. T. Hood

91-17088

SUMMARY

Decompiling is the process of deriving a computer program in a high-level language from one in machine-code or assembly language. Defence applications of decompiling include maintenance of obsolescent equipment, production of scientific and technical intelligence and assessment of systems for hazards to safety or security. This paper describes an approach to the rapid generation of decompilers through the use of Definite Clause Grammars., a class of abstract grammars which can be executed as Prolog programs. The approach is illustrated using "toy" languages. An environment which permits the integration of diverse sources of knowledge relevant to the decompilation problem and provides a graphical interface is described.

COPY NO. 43

Sept 91

POSTAL ADDRESS: Director, Electronics Research Laboratory,PO Box 1600, Salisbury, South Australia, 5108. ERL-0571-RR

91 12 4 084

CONTENTS

FIGURES

1 INTRODUCTION

The emerging discipline of software engineering envisages computer software being developed from a statement of requirements, through several stages of formal specification, coding and testing. There are, however, situations which demand the assessment and possible modification of the final products of the development process, for example executable machine code, in the absence of any other descriptions of the system. The process of (re)creating higher level, that is, more abstract, descriptions of the system, which may only have existed in the original designer's mind, is called reverse engineering.

Recent references to the need for reverse engineering of software include bringing large bodies of existing code under the umbrella of computer-aided software engineering (CASE) systems [Bachman 1988]. For many enterprises, the body of existing code represents a large investment and may embody corporate knowledge not recorded elsewhere. This has spawned a significant industry providing tools and contracted expertise supporting activities such as the transformation of "spaghetti code" into well-structured programs and the translation of programs in old languages into ones which are supported on modern systems [Kotik and Markosian 1989]. While some of this work has been concerned with assembly languages, for example on IBM mainframes, most has concerned higher-level languages such as FORTRAN.

A problem closer to the subject of this paper, namely the recovery of a higher-level language program from executable machine code, was that tackled by several groups in the United States in determining the behaviour and structure of the notorious "Internet Worm" program [Spafford 1988]. This work was done without the aid of any automated tools, apart from the use of the UNIX C compiler for checking hypotheses (Eugene Spafford, private communication).

Defence, with the longevity of its equipment, non-standard embedded processors and requirements for rapid modifications in response to new threats, countermeasures or operating environments, has a particular interest in reverse engineering tools and techniques. Reverse engineering of software is also relevant to the production of scientific and technical intelligence and to the assessment of otherwise "black box" systems for hazards to safety or security. The requirements for quick reaction and secrecy raised by many of these applications argues for powerful tools which can be rapidly customised to suit the problem at hand and which permit the job to be completed by a small number of analysts.

We also note, without comment, the increasingly common practice of proscribing reverse engineering of licensed software, exemplified in the following quotation from the "License and limited warranty agreement" printed in the reference manual for a personal computer electronic mail system: *"You may not . . . reverse engineer, disassemble, decompile, or make any attempt to discover the source code to the software"* [CE Software 1989].

Decompiling is the process of transforming a program expressed in assembly language or machine code into a description in a high-level language such as Ada or C which, with the aid of a suitable compiler, can be transformed into the original code. This paper describes some experiments in using the language Prolog for the rapid construction of decompilers. Familiarity with Prolog notation is assumed and the reader is referred to a textbook such as Sterling and Shapiro [1986]. Sufficient Prolog code is included to permit the reader to experiment with and extend the examples discussed.

2. LANGUAGE PROCESSING WITH PROLOG

Prolog was originally developed as a tool for implementing natural-language understanding systems [Colmerauer 1975]. The reader is referred to the paper by Pereira and Warren [1980] or any Prolog textbook [Sterling and Shapiro 1986] for an introduction to the topic. In general, the construction of useful natural-language processing systems is still largely a research activity, with Prolog being the tool of choice for some major current projects [Alshawi, Moore, Moran and Pulman 1988]. In general, programming languages are considerably simpler than natural languages, so that the construction of compilers in Prolog is quite straightforward [Warren 1980, Cohen and Hickey 1987].

Most programming languages produced after Algol-60 have their syntax defined by a formal context-free grammar, normally expressed in a notation called Backus-Naur Form (BNF) [Aho and Ulman 1977]. The power of Prolog for language processing is conveyed by the fact that it allows a notational-variant of BNF, called "grammar rules", to be simply transformed into a Prolog program which, when executed, accepts syntactically correct programs (or, in general, "sentences"). Prolog grammar rules actually define an extension of context-free grammars called definite-clause grammars (DCGs), with the descriptive power of a general purpose computer. In practice, most commercial compilers of Prolog are themselves written in Prolog.

Grammar rules have a single Prolog term, representing a non-terminal symbol of the grammar on the left hand side of the arrow, while the right-hand side contains terms representing other non-terminals, Prolog lists representing sequences of terminal symbols and arbitrary Prolog code enclosed in braces used to apply constraints or implement side-effects. Grammar rules are normally translated into executable Prolog by augmenting non-terminal symbols with additional arguments representing the input list of symbols and the list following the recognised symbol, while terminal symbols are translated into a format wherein they appear at the head of the input list [Pereira and Warren 1980]. Prolog code enclosed in braces is unchanged in translation. Most Prolog interpreters or compilers recognise and translate grammar-rules interspersed with clauses in standard notation. Others may provide a library procedure for the purpose. For example, the rule:

```
nonterminal_1(Attribute) -->
      nonterminal_2(Attribute), [terminal], {constraint(Attribute)}.,
```

which may be read as:

"*nonterminal_1(Attribute)* can replace *nonterminal_2(Attribute)* followed by the symbol '*terminal*', provided that *constraint(Attribute)* is satisfied",

might be translated into:

```
nonterminal_1(Attribute,X,Y) :-
    nonterminal_2(Attribute,X,Z),
    'C'(Z,terminal,Y),
    constraint(Attribute).
```

where the system predicate 'C' (for "connects") is defined by:

```
'C'([X|S],X,S).
```

DCGs are executed top-down much like a recursive-descent parser for a context-free grammar. The above rule would be invoked with variable X bound to a list of symbols. If the rule is successfully applied, variable Y would become bound to the list of symbols following those subsumed by *nonterminal_1(Attribute)*.

2.1 A Toy Compiler

Sterling and Shapiro [1986] provide the complete Prolog code implementing a multi-pass compiler for a toy language, PL, with a syntax similar to a subset of Pascal [Jensen and Wirth 1975] into a fictitious machine instruction set. Their example is based on one due to Warren [1980]. Their machine instruction set provides a single accumulator with both immediate and direct memory addressing and conditional branch instructions. The compiler of Sterling and Shapiro has been extended to accept source code from a file and to generate assembly code formatted for the convenience of the decompiler, as explained below. The compiler comprises a tokeniser, code-generator and assembler, in addition to the DCG parser shown in Figure 1. Figure 2 shows an example of the input and Figure 3 the invocation of the compiler with the resulting absolute machine code and symbol table.

2.2 A Toy Decompiler

Figure 4 shows a DCG parser which accepts the output of the above compiler, building in the process a description of the software as a Prolog term containing Pascal-like control structures. The non-terminals associated with arithmetic operations (`arith_exp` and `arith_op`) take as their third arguments terms of the form:

```
X => Y.
```

where `X` and `Y` unify with the content of the accumulator before and after the relevant operation respectively. This notation is readily extended to a more complex processor by providing arguments to represent additional registers or flags. The information needed to construct the decompiler was obtained through inspection of the compiler.

The parsing of standard control structures often requires the recognition of a jump to the next instruction following the parsed sequence. While DCGs are quite capable of performing calculations on addresses, it is more elegant to incorporate in each assembler instruction its address and the address of the location following the instruction. This allows control structures to be recognized using only unification (so that symbolic labels can be used if desired) and permits instructions of varying lengths. The program requires its input in the form of a Prolog list. Producing the required format from standard assembly listings would be a trivial task using Prolog or standard UNIX data-manipulation tools [Bourne 1983].

Figure 5 shows the structure built by this "decompiler" when given the code from Figure 3 and the results of formatting this according to the syntax of PL using a simple pretty-printer. This output can be successfully recompiled. The decompiler handles all of the constructs allowed in the PL language.

```
/*
      parse(Tokens,Structure) :-
            Structure represents the successfully parsed list
            of Tokens.
*/

parse(Source,Structure) :-
      pl_program(Structure, Source, []).

pl_program(S) --> [program], identifier(X), [';'], statement(S).

statement((S;Ss)) -->
    [begin], statement(S), rest_statements(Ss).
statement(assign(X,V)) -->
    identifier(X), [':='], expression(V).
statement(if(T,S1,S2)) -->
    [if], test(T), [then], statement(S1), [else], statement(S2).
statement(while(T,S)) -->
    [while], test(T), [do], statement(S).
statement(read(X)) -->
    [read], identifier(X).
statement(write(X)) -->
    [write], expression(X).

rest_statements((S;Ss)) --> [';'], statement(S), rest_statements(Ss).
rest_statements(void) --> [end].

expression(X) --> pl_constant(X).
expression(expr(Op,X,Y)) -->
      pl_constant(X), arithmetic_op(Op), expression(Y).

arithmetic_op('+') --> ['+'].
arithmetic_op('-') --> ['-'].
arithmetic_op('*') --> ['*'].
arithmetic_op('/') --> ['/'].

pl_constant(name(X)) --> identifier(X).
pl_constant(number(X)) --> pl_integer(X).

identifier(X) --> [X], {atom(X)}.
pl_integer(X) --> [X], {integer(X)}.

test(compare(Op,X,Y)) -->
      expression(X), comparison_op(Op), expression(Y).

comparison_op('=') --> ['='].
comparison_op('>') --> ['>'].
comparison_op('<') --> ['<'].
comparison_op('>=') --> ['>='].
comparison_op('=<') --> ['=<'].
```

Figure 1. A DCG parser for the PL language (Sterling and Shapiro 1986).

```
program factorial;
    begin
         read value;
         count  :=  1;
         result  :=  1;
         while count  <  value do
               begin
                    count  :=  count  +  1;
                    result  :=  result  *  count
               end;
         write result
     end.
```

Figure 2. A PL program

```
?- cfile(factorial_pl).

[(value,18),(count,19),(result,20)]

[
[0,1,read,18],
[1,2,loadc,1],
[2,3,store,19],
[3,4,loadc,1],
[4,5,store,20],
[5,6,load,19],
[6,7,sub,18],
[7,8,jumpge,15],
[8,9,load,19],
[9,10,addc,1],
[10,11,store,19],
[11,12,load,20],
[12,13,mul,19],
[13,14,store,20],
[14,15,jump,5],
[15,16,load,20],
[16,17,write,0],
[17,18,halt,0]
].  yes
?-
```

Figure 3 Script of a Prolog session (user input in italics) showing the invocation of the PL compiler with the file factorial_pl containing the code shown in Figure 2. The output comprises the symbol table followed by the assembled object code.

```
?- op(700, xfx, ':='), op(700, yfx, (=>)), op(900, fy, not), op(100, fx, var).

decomp(Code, PL) :- pl_prog(0, _, PL, Code, []).

pl_prog(P, Q, prog(X)) --> pl_frag(P, P1, X), [[P1, Q, halt, 0]].

pl_frag(P, Q, X) --> stmt(P, Q, X).
pl_frag(P, R, (X;Y)) --> stmt(P, Q, X), pl_frag(Q, R, Y).

stmt(P, Q, read(var(A))) --> [[P, Q, read, A]].
stmt(P, Q, write(var(A))) -->
        [[P, P1, load, A], [P1, Q, write, 0]].
stmt(P, Q, noop) --> [[P, Q, noop, _]].
stmt(P, Q, while(Test, Do)) -->
        branch_if_not(_, _, Test, Q),
        pl_frag(_, _, Do),
        [[P1, Q, jump, P]].
stmt(P, Q, if(Test, Then, Else)) -->
        branch_if_not(_, _, Test, P1),
        pl_frag(_, _, Then),
        [[_, _, jump, Q]],
        pl_frag(P1, Q, Else).
stmt(P, Q, if(Test, Then)) -->
        branch_if_not(_, _, Test, Q),
        pl_frag(_, Q, Then).
stmt(P, Q, Asgns) -->
        arith_exp(P, P1, _ => X), assign_seq(P1, Q, Asgns, X).

assign_seq(P, Q, var(A) := B, X) -->
        [[P, P1, store, A]], assign_seq(P1, Q, B, X).
assign_seq(P, Q, var(A) := X, X) --> [[P, Q, store, A]].

branch_if_not(P, Q, Test, R) -->
        arith_exp(P, _, _ => A), arith_op(_, _, A => A-B),
        { comparison_opcode(Comp, JumpOp), func_args(Test, Comp, A, B) },
         [Q, _, JumpOp, R]].

arith_exp(P, Q, A) --> arith_op(P, Q, A).
arith_exp(P, Q, A => C) -->
        arith_op(P, P1, A => B), arith_exp(P1, Q, B => C).

arith_op(P, Q, _ => A) --> [[P, Q, loadc, A]].
arith_op(P, Q, _ => var(A)) --> [[P, Q, load, A]].
arith_op(P, Q, A => E) -->
         [P, Q, Op, B]],
        { literal_operation(Sym, Op), func_args(E, Sym, A, B) }.
arith_op(P, Q, A => E) -->
        [[P, Q, Op, B]],
        { memory_operation(Sym, Op), func_args(E, Sym, A, var(B)) }.

comparison_opcode('=',jumpne). comparison_opcode('>',jumple).
comparison_opcode('>=',jumplt). comparison_opcode('<',jumpge).
comparison_opcode('=<',jumpgt).

literal_operation('+',addc). literal_operation('-',subc).
literal_operation('*',mulc). literal_operation('/',divc).

memory_operation('+',add). memory_operation('-',sub).
memory_operation('*',mul). memory_operation('/',div).

% access components of Term = Functor(Arg1, Arg2) (more efficient than =.. )
func_args(Term, Functor, Arg1, Arg2) :-
        functor(Fact, Functor, 2),
        arg(1, Fact, Arg1),
        arg(2, Fact, Arg2).
```

Figure 4 A DCG parser for decompiling the output of the PL compiler

```
prog((read(var 18);
      (var 19 := 1;
       (var 20 := 1;
        (while(var 19 < var 18,
               (var 19 := var 19 + 1;
                var 20 := var 20 * var 19));
         write(var 20))))))
```

(a) **The structure created during the decompilation of code of Figure 3 using the grammar of Figure 4.**

```
program thing;
begin
    read var18;
    var19 := 1;
    var20 := 1;
    while var19<var18 do begin
        var19 := var19+1;
        var20 := var20*var19
    end ;
    write var20
end.
```

(b) **The above term pretty-printed in a format acceptable to the PL compiler.**

Figure 5

2.3 Compiler Optimisations

One of the problems real decompilers will face is the handling of compiler optimisations. This is illustrated by a simple case which is handled by our toy decompiler. The compiler described above translates the PL sequence:

```
a := b;
c := b
```

into the assembler sequence:

```
load b
store a
load b
store c
```

An optimising compiler would recognise that the second *load* is redundant and would remove it. The rule for recognising assignment statements in the decompiler accepts the "optimised" code:

```
load b
store a
store c
```

and translates it into the form:

```
c := a := b
```

which could, if desired, be transformed into two separate assignments to match PL syntax rules.

A full discussion of the decompilation of optimised code is beyond the scope of this paper.

3. DECOMPILING "SMALL-C" FOR THE INTEL 8085

The instruction set used in the above example is, unfortunately, rather different from those of typical microprocessors. In order to provide a more realistic evaluation, the Intel 8085 8-bit microprocessor [Intel 1977] was chosen. A convenient high-level language was provided by the public domain Small-C compiler,, which accepts a large subset of the C language [Kernighan and Ritchie 1978]. We describe below the construction of decompilers for two subsets of Small-C including *while* and *if-then-else* control structures and assignment statements with arithmetic expressions. The first employs only *static* integer variables (variables are assigned addresses in memory), the second only *automatic* variables (variables are assigned on the system stack so that storage for them is created and destroyed on procedure entry and exit). A full decompiler for the language would include rules for both classes of variables as well as character and pointer data-types, more complex expressions and the remaining control structures.

3.1 Static Variables

Figure 6 shows a C version of the factorial program (without the read and write commands for simplicity) and a fragment of the 8085 assembly language generated by the Small-C compiler. One noticeable difference from the toy instruction set is that the *store* step in an assignment statement is now delocalised, with the address calculated and *pushed* onto the stack prior to evaluation of the expression whence it is subsequently *popped* for use.

Figure 8 shows the input and output. The assembly language was formatted manually using a text editor. Arbitrary numerical addresses, which are only used for unification so they could be any unique symbols, have been used in the address fields, while labels for variable locations and run-time routines have been left in their original form by declaring "?" as a prefix operator. It is, perhaps, interesting to note that Pascal-like code has been produced by decompiling the output of a C compiler (although the subset of C we have chosen is easily mapped into Pascal).

```
main()
{
        static int value, count, result;
        value = 10;
        count = 1;
        result = 1;
        while (count < value) {
                count = count + 1;
                result = result * count;
        }
}
```

(a) A Small-C program using static variables

```
;       Small C 8080;
;       Coder (2.4,84/11/27)
;       Front End (2.7,84/11/28)
        extern  ?pint
           ...
        cseg
main:

      ; Allocate storage for variables in data segment.

        dseg
?2:     ds      2
        cseg
        dseg
?3:     ds      2
        cseg
        dseg
?4:     ds      2
        cseg

      ; Load HL register-pair with address of variable.
      ; and save it on the stack.

        lxi     h,?2
        push    h

      ; Load HL register-pair with value 10.

        lxi     h,10

      ; Load DE register-pair with address of variable
      ; and call run-time routine to store an integer.

        pop     d
        call    ?pint
      ...
```

(b) 8085 assembly language produced by the Small-C compiler from the program of (a), up to the end of the first assignment statement (count = 10). Added comments are shown in italics.

Figure 6

Figure 7 shows the grammar rules of the decompiler. In this implementation of C, the HL register assumes the role of accumulator in arithmetic operations.

```
decomp(Code, PL) :- pl_prog(_, _, PL, Code, []).

pl_prog(P, Q, prog(X)) --> pl_frag(P, P1, X), [P1, Q, ret].

pl_frag(P, Q, X) --> stmt(P, Q, X).
pl_frag(P, R, (X;Y)) --> stmt(P, Q, X), pl_frag(Q, R, Y).

stmt(P, Q, while(Test, Do)) -->
        branch_if_not(P, P1, Test, Q),
        pl_frag(P1, P2, Do),
         [P2, Q, jmp, P].
stmt(P, Q, if(Test, Then, Else)) -->
        branch_if_not(P, P1, Test, P3),
        pl_frag(P1, P2, Then),
         [P2, P3, jmp, Q],
        pl_frag(P3, Q, Else).
stmt(P, Q, if(Test, Then)) -->
        branch_if_not(P, P1, Test, Q),
        pl_frag(P1, Q, Then).
stmt(P, Q, var(A) := X) -->
        [P, P1, lxi, h, A],
         [P1, P2, push, h],
        arith_exp(P2, P3, _ => X),
         [P3, P4, pop, d],
         [P4, Q, call, ?pint].

branch_if_not(P, Q, Test, R) -->
        arith_exp(P, P1, _ => A),
         [P1, P2, push, h],
        arith_exp(P2, P3, _ => B),
         [P3, P4, pop, d], [P4, P5, call, Subr],
        { comp_op(Comp, Subr), func_args(Test, Comp, A, B) },
         [P5, P6, mov, a, h], [P6, P7, ora, l], [P7, Q, jz, R].

comp_op('=', ?eq). comp_op('<', ?lt).
comp_op('>', ?gt). comp_op('=<', ?le).
comp_op('>=', ?ge). comp_op('!=', ?ne).

arith_exp(P, Q, A) --> arith_op(P, Q, A).
arith_exp(P, Q, A => C) -->
        arith_op(P, P1, A => B), arith_exp(P1, Q, B => C).

arith_op(P, Q, _ => A) --> [[P, Q, lxi, h, A]].
arith_op(P, Q, _ => var(A)) -->
         [P, _, lxi, h, A], [_, Q, call, ?gint]].
arith_op(P, Q, A => E) -->
  [P, _, push, h]],
         arith_exp(_, _, _ => B),
         [_, _, pop, d]],
        do_arith_op(_, Q, Op),
        {func_args(E, Op, A, B)}.

do_arith_op(P, Q, '+') --> [[P, Q, dad, d]].
do_arith_op(P, Q, '-') --> [[P, Q, call, ?sub]].
do_arith_op(P, Q, '*') --> [[P, Q, call, ?mul]].
do_arith_op(P, Q, '/') --> [[P, Q, call, ?div]].
```

Figure 7. A decompiler which accepts 8085 assembly language from a subset of the Small-C language using only *static* variables.

```
example([
         [1, 2,       lxi,     h,?2],
         [2, 3,       push,    h],
         [3, 4,       lxi,     h,10],
         [4, 5,       pop,     d],
         [5, 6,       call,    ?pint],
         [6, 7,       lxi,     h,?3],
         [7, 8,       push,    h],
         [8, 9,       lxi,     h,1],
         [9, 10,      pop,     d],
         [10, 11,     call,    ?pint],
         [11, 12,     lxi,     h,?4],
         [12, 13,     push,    h],
         [13, 14,     lxi,     h,1],
         [14, 15,     pop,     d],
         [15, ?5,     call,    ?pint],
         [?5, 17,     lxi,     h,?3],
         [17, 18,     call,    ?gint],
         [18, 19,     push,    h],
         [19, 20,     lxi,     h,?2],
         [20, 21,     call,    ?gint],
         [21, 22,     pop,     d],
         [22, 23,     call,    ?lt],
         [23, 24,     mov,     a,h],
         [24, 25,     ora,     l],
         [25, 26,     jz,      ?6],
         [26, 27,     lxi,     h,?3],
         [27, 28,     push,    h],
         [28, 29,     lxi,     h,?3],
         [29, 30,     call,    ?gint],
         [30, 31,     push,    h],
         [31, 32,     lxi,     h,1],
         [32, 33,     pop,     d],
         [33, 34,     dad,     d],
         [34, 35,     pop,     d],
         [35, 36,     call,    ?pint],
         [36, 37,     lxi,     h,?4],
         [37, 38,     push,    h],
         [38, 39,     lxi,     h,?4],
         [39, 40,     call,    ?gint],
         [40, 41,     push,    h],
         [41, 42,     lxi,     h,?3],
         [42, 43,     call,    ?gint],
         [43, 44,     pop,     d],
         [44, 45,     call,    ?mul],
         [45, 46,     pop,     d],
         [46, 47,     call,    ?pint],
         [47, ?6,     jmp,     ?5],
         [?6, 48,     ret] ]).
```

(a) 8085 assembly language from the compilation of the Small-C program of Figure 6. formatted for decompiling using the program of Figure 7.

```
program thing;
begin
    var(?2) := 10;
    var(?3) := 1;
    var(?4) := 1;
    while var(?3)<var(?2) do begin
        var(?3) := var(?3)+1;
        var(?4) := var(?4)*var(?3)
    end
end.
```

(b) Formatted output generated by the decompiler of Figure 7 from (a).

Figure 8

3.2 Automatic Variables

Automatic variables are accessed by calculating an offset from the current value of the stack pointer (a register known as SP in the case of the 8085). The decompiler grammar is complicated by the need to account for changes in SP as the stack is used for temporary storage during expression evaluation, as shown in the annotated assembly code in Figure 9. It is apparent that the number of *pushes* during a C statement is balanced by the number of *pops*, so it is sufficient to include an extra variable in the non-terminals used in arithmetic expressions (Figure 10) to record SP decrements (by 2 with each push) to decide which variable is being accessed.

```
;       Small C 8080;
;       Coder (2.4,84/11/27)
;       Front End (2.7,84/11/28)
        extern  ?pint
      ...

        cseg
main:

      ; Establish a stack frame for 3 integer variables.

        push    b
        push    b
        push    b

      ; Calculate variable address as offset from current
      ; stack-pointer (SP) and save it on the stack.

        lxi     h,4
        dad     sp
        push    h

      ; Load HL register-pair with value 10

        lxi     h,10

      ; Load DE register-pair with address of variable
      ; and call run-time routine to store an integer.

        pop     d
        call    ?pint
      ...
```

Figure 9 8085 assembly language produced by the Small-C compiler from the program of Figure 6a, (but with variables declared *int*, rather than static *int*) up to the end of the first assignment statement (count = 10). Added comments are shown in italics.

Formatting the code of Figure 9 as a Prolog list in the manner of Figure 8a, and employing the grammar of Figure 10 in the decompiler, we obtain a listing identical (apart from the names of variables) to that of Figure 8b.

```
decomp(Code, PL) :-
        d_prog(_, _, PL, Code, []).

d_prog(P, Q, prog(X)) -->
        pushes(P, P1, 0, N),
        p_frag(P, P1, X),
        pops(P, P1, 0, N),
         [[P1, Q, ret]],
        {write(num_of_vars = N), nl}.

p_frag(P, Q, X) --> stmt(P, Q, X).
p_frag(P, R, (X;Y)) -->
        stmt(P, Q, X),
        p_frag(Q, R, Y).

stmt(P, Q, while(Test, Do)) -->
        branch_if_not(P, P1, Test, Q),
        p_frag(P1, P2, Do),
        [[P2, Q, jmp, P]].
stmt(P, Q, if(Test, Then, Else)) -->
        branch_if_not(P, P1, Test, P3),
        p_frag(P1, P2, Then),
         [[P2, P3, jmp, Q]],
        p_frag(P3, Q, Else).
stmt(P, Q, if(Test, Then)) -->
        branch_if_not(P, P1, Test, Q),
        p_frag(P1, Q, Then).
stmt(P, Q, var(A) := X) -->
         [[P, P1, lxi, h, A],
          [P1, P2, dad, sp],
          [P2, P3, push, h]],
        arith_exp(P3, P4, _ => X, 2),
         [[P4, P5, pop, d],
          [P5, Q, call, ?pint]].

branch_if_not(P, Q, Test, R) -->
        arith_exp(P, P1, _ => A, 0),
        [[P1, P2, push, h]],
        arith_exp(P2, P3, _ => B, 2),
        [[P3, P4, pop, d],
         [P4, P5, call, Subr]],
        {
          comp_op(Comp, Subr),
          func_args(Test, Comp, A, B)
        },
        [[P5, P6, mov, a, h],
         [P6, P7, ora, l],
         [P7, Q, jz, R]].
```

```
comp_op('=', ?eq). comp_op('<', ?lt).
comp_op('>', ?gt). comp_op('=<', ?le).
comp_op('>=', ?ge). comp_op('\=', ?ne).

arith_exp(P, Q, A, S) -->
        arith_op(P, Q, A, S).
arith_exp(P, Q, A => C, S) -->
        arith_op(P, P1, A => B, S),
        arith_exp(P1, Q, B => C, S).

arith_op(P, Q, _ => A, S) -->
        [[P, Q, lxi, h, A]].
arith_op(P, Q, _ => var(A), S) -->
        [[P, _, lxi, h, X],
         [_, _, dad, sp],
         [_, Q, call, ?gint]],
        {plus(A, S, X)}.
arith_op(P, Q, A => E, S) -->
        [[P, _, push, h]],
        {plus(S, 2, S1)},
        arith_exp(_, _, _ => B, S1),
        [[_, _, pop, d]],
        do_arith_op(_, Q, Op),
        {func_args(E, Op, A, B)}.

do_arith_op(P, Q, '+') -->
        [[P, Q, dad, d]].
do_arith_op(P, Q, '-') -->
        [[P, Q, call, ?sub]].
do_arith_op(P, Q, '*') -->
        [[P, Q, call, ?mul]].
do_arith_op(P, Q, '/') -->
        [[P, Q, call, ?div]].

pushes(P, Q, N0, N) -->
        [[P, P1, push, b]],
        pushes(P1, Q, N0, N1),
        {plus(N1, 1, N)}.
pushes(P, Q, N, N) --> [].

pops(P, Q, N0, N) -->
        [[P, P1, pop, b]],
        pops(P1, Q, N0, N1),
        {plus(N1, 1, N)}.
pops(P, Q, N, N) --> [].
```

Figure 10 A "decompiler" for 8080 assembly language generated by the "SmallC" compiler using only automatic variables. Decompilation results in a "generic" block-structured representation which can be formatted to produce a language of choice (eg Pascal).

4. INTERACTIVE DECOMPILING

4.1 Using the Prolog Database

The standard approach to translating DCG rules into executable Prolog, which has been used in the examples thus far, requires the input assembler code to be represented as a Prolog list. While this normally provides faster execution than other approaches [Pereira and Warren 1980], it has some disadvantages.

We have not considered here many of the processes which might be required prior to decompilation, such as the separation of code from data [Horspool and Marovac 1980], or the determination of the semantics of run-time procedures not part of the available code. It is suggested that the reverse-engineering of a large program will require significant interaction with a human analyst over a considerable period and that more rapid progress will be made on some sections of the code than on others.

The reasoning applied by a human analyst might be sometimes bottom-up, or data-driven:

"that small procedure has two 8-bit XOR instructions; it probably is doing a 16-bit XOR",

and at other times top-down or hypothesis-driven:

"let's assume this was written in PL/M-80".

The model of the reverse-engineering process which we have in mind appears to be a close fit to the so-called Blackboard Model which had its origins in computer understanding of speech [Nii 1986A, 1986B].

Pereira and Warren [1980] note that lists are not the only way of representing sequences of symbols in implementing DCG parsers. In particular, if individual instructions are stored as facts in the Prolog database, any rule of a decompiler grammar to be applied starting at any instruction, allowing a combination of top-down and bottom-up parsing. Further, as pointed out by Pereira and Warren, when a rule has been successfully applied we can add a fact containing the recognised structure to the database so that it can be considered by other rules without repeating the computation required in its recognition. Parsers which employ such tables (or charts) of already recognised well-formed substrings are often called chart-parsers [Winograd 1983].

Figure 11 is an interpreter for a DCG grammar which expects input symbols to appear as facts in a ternary relation *e* (standing for "edge": chart-parsing jargon), referred to here as the *chart*. In order to avoid confusion, a different arrow symbol has been used in the grammar-rules. Note that this interpreter inspects the chart for a required structure before looking for a rule. Recognised PL statements are added to the database, but smaller fragments are not. This appears to be a reasonable approach, given the relative expense of the assert operation. These assertions of deduced results into the database are within the spirit of logic programming as they do not change the meaning of the program.

```
substr(P, Q, (A, As)) :-
      substr(P, P1, A),
      substr(P1, Q, As).
substr(P, P, []).
substr(P, P, (X!) :- X, !.
substr(P, Q, e(P, Q, T)) :- substr(P, Q, T).
substr(P, Q, T) :- e(P, Q, T), !.
substr(P, Q, T) :-
            (T ==> A),
            substr(P, Q, A),
            (T = stmt(S) -> new_edge(e(P, Q, T)) ; true).

new_edge(E) :- E, !.
new_edge(E) :-
      asserta(E),
      write(E),
      nl.
```

Figure 11 Chart-parser implemented as an interpreter of DCGs.

The DCGs employed in the previous examples required each non-terminal contain parameters representing the current and next "address". As this is information is now included in the chart, it need only appear in rules when it is necessary for recognising control structures. In such cases, direct reference is made to the chart representation. Generally rules are less cluttered than in the previous notation.

Figure 12 shows the grammar of Figure 10 in the new format. Figure 13 shows the assembly code in the format for use with the chart parser. Note that, as with the list-based representation, the arguments `(I, J)` which link successive facts `e(I, J, Instruction)` are used only for unification so they can be any unique terms. Figure 14 shows a script of a Prolog session in an environment containing the clauses of Figures 11, 12, 4b and 13. Running this example (with the printing of messages disabled) on an Apple Macintosh-Plus under Advanced AI Systems Prolog requires 4.08 seconds of CPU time. If no recognised structures are saved in the chart, the time increases to 7.35 seconds. For comparison, the time taken for the equivalent problem using the list-based representation (Figure 10) is 2.1 seconds. Execution speed using the chart representation could be substantially improved by "compiling" grammar rules into equivalent Prolog clauses, as is done for the list-based representation. For our present purposes, the use of an interpreter facilitates experimentation.

```
pl_prog(prog(X)) ==>
        pushes(0, N),
        pl_frag(X),
        pops(0, N),
        [ret],
        {write('number of automatic variables' - N), nl}.

pl_frag(X) ==> stmt(X).
pl_frag((X;Y)) ==> stmt(X), pl_frag(Y).

stmt(while(Test, Do)) ==>
        e(P, _, branch_if_not(Test, Q)),
        pl_frag(Do),
        e(_, Q, [jmp, P]).
stmt(if(Test, Then, Else)) ==>
        branch_if_not(Test, P),
        pl_frag(Then),
        [jmp, Q],
        e(P, Q, pl_frag(P3, Q, Else)).
stmt(if(Test, Then)) ==>
        branch_if_not(Test, Q),
        e(_, Q, pl_frag(Then)).
stmt(var(A) := X) ==>
         [lxi, h, A], [dad, sp], [push, h],
        arith_exp(_ => X, 2),
         [pop, d], [call, ?pint].

branch_if_not(Test, R) ==>
        arith_exp(_ => A, 0),
        [push, h],
        arith_exp(_ => B, 2),
         [pop, d], [call, Subr],
        {
                comp_op(Comp, Subr),
                func_args(Test, Comp, A, B)
        },
        [mov, a, h], [ora, l], [jz, R].

comp_op('=', ?eq).
comp_op('<', ?lt).
comp_op('>', ?gt).
comp_op('=<', ?le).
comp_op('>=', ?ge).
comp_op('\=', ?ne).

arith_exp(A, S) ==> arith_op(A, S).
arith_exp(A => C, S) ==>
        arith_op(A => B, S), arith_exp(B => C, S).

arith_op(_ => A, S) ==> [lxi, h, A].
arith_op(_ => var(A), S) ==>
        [lxi, h, X], [dad, sp], [call, ?gint],
        {plus(A, S, X)}.
arith_op(A => E, S) ==>
        [push, h],
        {plus(S, 2, S1)},
        arith_exp( _ => B, S1),
        [pop, d],
        do_arith_op(Op),
        {func_args(E, Op, A, B)}.

do_arith_op('+') ==> [dad, d].
do_arith_op('-') ==> [call, ?sub].
do_arith_op('*') ==> [call, ?mul].
do_arith_op('/') ==> [call, ?div].

pushes(N0, N) ==> [push, b], pushes(N0, N1), {plus(N1, 1, N)}.
pushes(N, N) ==> [].

pops(N0, N) ==> [pop, b], pops(N0, N1), {plus(N1, 1, N)}.
pops(N, N) ==> [].
```

Figure 12 The grammar of Figure 10 re-cast for use with the chart-parsing interpreter of Figure 11.

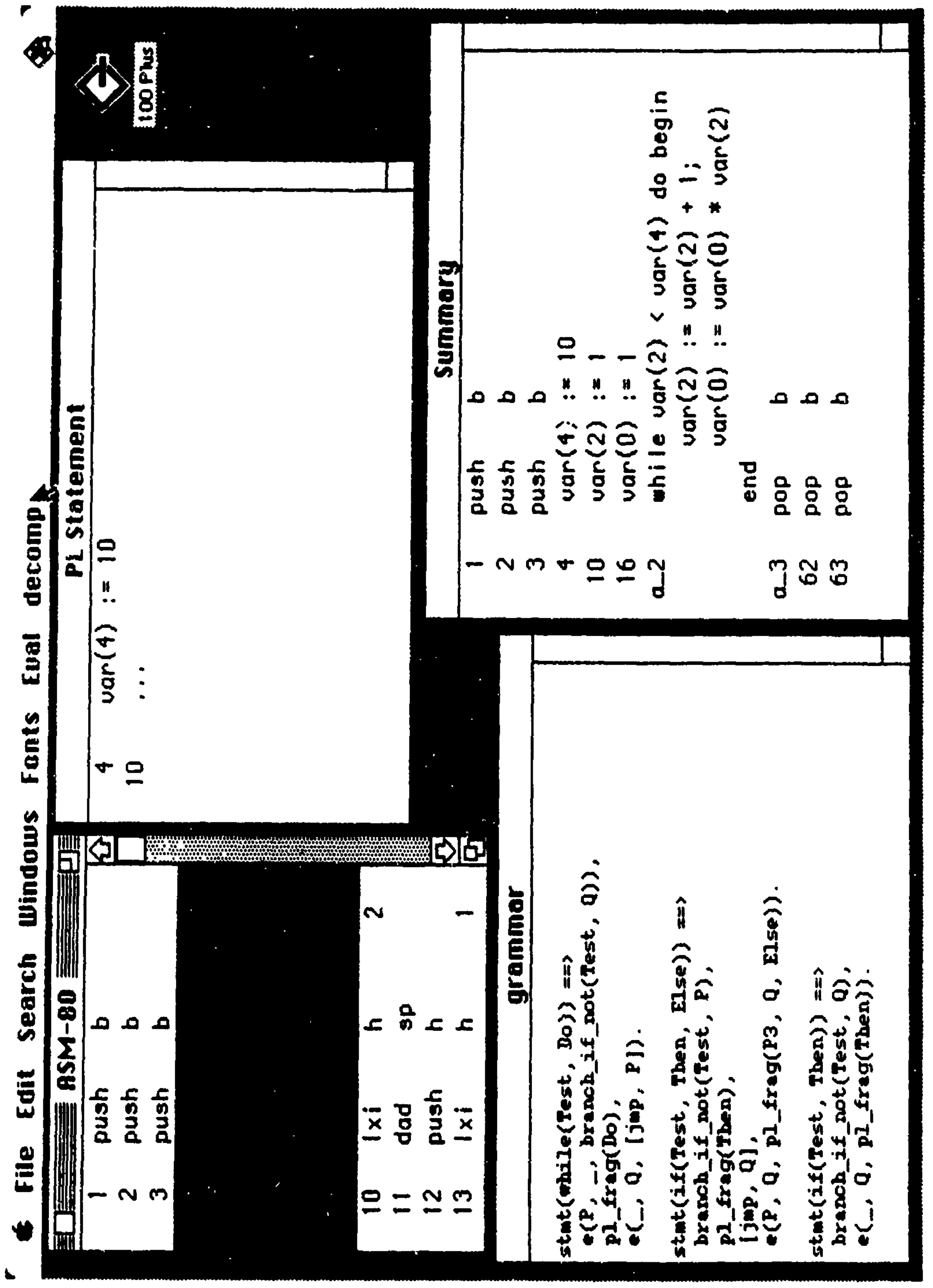
```
e(1, 2,     [push,    b]).          e(33, 34,    [jz,      ?3]).
e(2, 3,     [push,    b]).          e(34, 35,    [lxi,     h,2]).
e(3, 4,     [push,    b]).          e(35, 36,    [dad,     sp]).
e(4. 5,     [lxi,     h,4]).        e(36, 37,    [push,    h]).
e(5, 6,     [dad,     sp]).         e(37, 38,    [lxi,     h,4]).
e(6, 7,     [push,    h]).          e(38, 39,    [dad,     sp]).
e(7, 8,     [lxi,     h,10]).       e(39, 40,    [call,    ?gint]).
e(8, 9,     [pop ,    d]).          e(40, 41,    [push,    h]).
e(9, 10,    [call,    ?pint]).      e(41, 42,    [lxi,     h,1]).
e(10, 11,   [lxi,     h,2]).        e(42, 43,    [pop,     d]).
e(11, 12,   [dad,     sp]).         e(43, 44,    [dad,     d]).
e(12, 13,   [push,    h]).          e(44, 45,    [pop,     d]).
e(13, 14,   [lxi,     h,1]).        e(45, 46,    [call,    ?pint]).
e(14, 15,   [pop,     d]).          e(46, 47,    [lxi,     h,0]).
e(15, 16,   [call,    ?pint]).      e(47, 48,    [dad,     sp]).
e(16, 17,   [lxi,     h,0]).        e(48, 49,    [push,    h]).
e(17, 18,   [dad,     sp]).         e(49, 50,    [lxi,     h,2]).
e(18, 19,   [push,    h]).          e(50, 51,    [dad,     sp]).
e(19, 20,   [lxi,     h,1]).        e(51, 52,    [call,    ?gint]).
e(20, 21,   [pop,     d]).          e(52, 53,    [push,    h]).
e(21, ?2,   [call,    ?pint]).      e(53, 54,    [lxi,     h,6]).
e(?2, 23,   [lxi,     h,2]).        e(54, 55,    [dad,     sp]).
e(23, 24,   [dad,     sp]).         e(55, 56,    [call,    ?gint]).
e(24, 25,   [call,    ?gint]).      e(56, 57,    [pop,     d]).
e(25, 26,   [push,    h]).          e(57, 58,    [call,    ?mul]).
e(26, 27,   [lxi,     h,6]).        e(58, 59,    [pop,     d]).
e(27, 28,   [dad,     sp]).         e(59, 60,    [call,    ?pint]).
e(28, 29,   [call,    ?gint]).      e(60, ?3,    [jmp,     ?2]).
e(29, 30,   [pop,     d]).          e(?3, 62,    [pop,     b]).
e(30, 31,   [call,    ?lt]).        e(62, 63,    [pop,     b]).
e(31, 32,   [mov,     a,h]).        e(63, 64,    [pop,     b]).
e(32, 33,   [ora,     l]).          e(64, 65,    [ret]).
```

Figure 13 8085 assembly instructions from the Small-C compiler formatted as prolog facts for application of the chart parser of Figure 11 and the grammar of Figure 12

```
?- substr(1, _, pl_prog(S)), pprint(S).
e(4,10,stmt(var4:=10))
e(10,16,stmt(var2:=1))
e(16,72,stmt(var0:=1))
e(34,46,stmt(var2:=var2+1))
e(46,60,stmt(var0:=var0*var2))
e(72,73,stmt(while(var2<var4,(var2:=var2+1;var0:=var0*var2))))
number of automatic variables=3
program thing;
begin
    var4 := 10;
    var2 := 1;
    var0 := 1;
    while var2<var4 do begin
        var2 := var2+1;
        var0 := var0*var2
    end
end.
    S = prog((var 4 := 10;
              (var 2 := 1;
               (var 0 := 1;
                while(var 2 < var 4,
                      (var 2 := var 2 + 1;
                       var 0 := var 0 * var 2))))))
?-
```

Figure 14 Script of a prolog session (user input in italics) showing the invocation of the chart-parser. Recognised structures are printed as they are asserted in the database.

Interactive decompilation would be greatly assisted by the analyst being able to indicate graphically the point in the assembly or machine code at which decompilation should start and, perhaps to select the program construct to be recognised from a menu. Figure 15 shows a prototype interactive decompilation environment written in Quintus MacProlog on an Apple Macintosh II computer. Apart from code concerned with the user interface, the Prolog program is that of the chart-parser described above. The window labelled "ASM-80" displays in conventional format the 8085 assembly language represented internally as in Figure 13. The window labelled "Grammar" displays the grammar of Figure 12 for browsing and, when appropriate, refinement and extension. The analyst has placed the cursor on the line labelled "4" to indicate the start point for an attempted decompilation and has selected "stmt(_)" (see Figure 12) as the syntactic category to be recognised. The successfully recognised PL structure, in this case an assignment statement, is displayed in the window labelled "PL Statement" and the assembly code which it spans is highlighted in the "ASM-80" window. The "Summary" window displays the current results of decompilation.



ASM-80

```
1   push   b
2   push   b
3   push   b

10  lxi    h    2
11  dad    sp
12  push   h
13  lxi    h    1
```

PL Statement

```
4   var(4) := 10
10  ...
```

Summary

```
1    push   b
2    push   b
3    push   b
4    var(4) := 10
10   var(2) := 1
16   var(0) := 1
a_2  while var(2) < var(4) do begin
         var(2) := var(2) + 1;
         var(0) := var(0) * var(2)
       end
a_3  pop    b
62   pop    b
63   pop    b
```

grammar

```
stmt(while(Test, Do)) ==>
 e(P, _, branch_if_not(Test, Q)),
 pl_frag(Do),
 e(_, Q, [jmp, P]).

stmt(if(Test, Then, Else)) ==>
 branch_if_not(Test, P),
 pl_frag(Then),
 [jmp, Q],
 e(P, Q, pl_frag(P3, Q, Else)).

stmt(if(Test, Then)) ==>
 branch_if_not(Test, Q),
 e(_, Q, pl_frag(Then)).
```

Figure 15 Interactive decompiling environment. See text for details.

4.2 Application Domain Semantics - The Symbol Table

The design of a computer program is only partially captured in the structure of its source code. In order to make it understandable, it must be related to the application domain. This is typically provided by the choice of meaningful names for variables and by the insertion of comments. The interactive environment proposed provides for this. Variables and labels derived during the decompilation can be assigned meaningful symbols by the analyst asserting facts of the form:

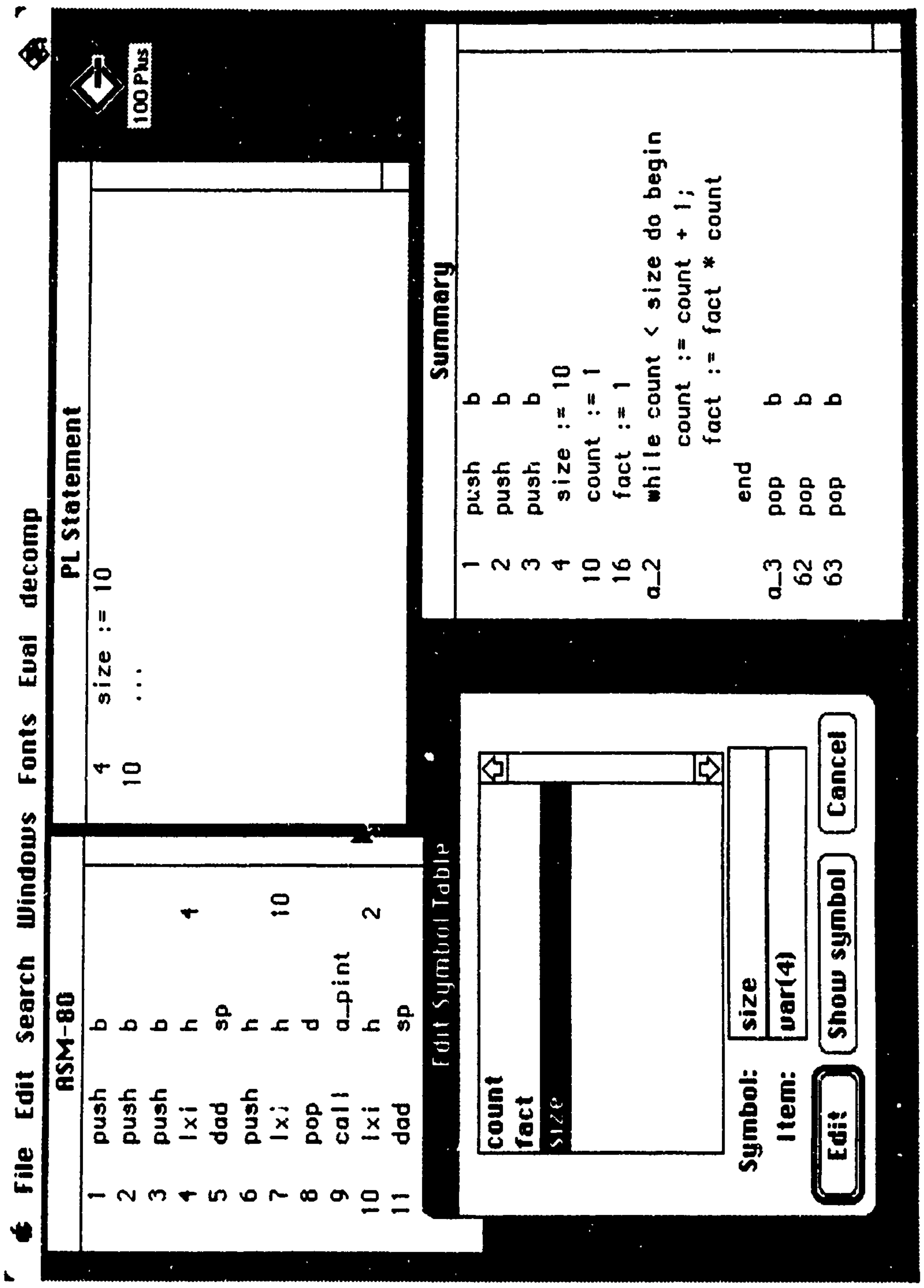
```
symbol(var(1), count).
symbol(1, start).
```

Comments can likewise be attached to segments of code by the analyst specifying the "address" range:

```
comment(1, 2, 'initialise count').
```

Many programming languages allow constants to be represented by symbolic names, with the actual value defined at one place in the code. This simplifies modification of the program as well as making it more easily understood. Replacement of the symbol with its numerical equivalent is a trivial operation for the compiler. The reversal of this process is in general difficult and requires understanding of the meaning of the program in the application domain. Unusual (so-called *magic numbers*) or recurring values might be brought to the attention of the analyst for possible replacement by symbolic constants. The recognition of constants with prosaic values which might also arise from many unrelated causes (for example the values 0 or 1) would require deep understanding of the program's domain semantics and is unlikely to be achieved automatically. A mechanism for replacing numerical values with symbols requires a means of referring to the values to be replaced. The simplest approach is to edit the assembly or machine code then re-run the decompiler over the modified structures.

The prototype interactive environment described above permits the user to assign symbolic names to variables. These are stored in a symbol table and used when PL code is displayed. Figure 16 shows the Macintosh "Dialog Box" for editing the symbol table. Arbitrary names for variables in the "Summary" window (Figure 15) have been replaced by (presumably) meaningful symbols. Currently a one-to-one correspondence between variables and symbols is required.



ASM-80

```
1   push  b
2   push  b
3   push  b
4   lxi   h      4
5   dad   sp
6   push  h
7   lxi   h      10
8   pop   d
9   call  a_pint
10  lxi   h      2
11  dad   sp
```

PL Statement

```
4    size := 10
10   ...
```

Summary

```
1    push  b
2    push  b
3    push  b
4    size := 10
10   count := 1
16   fact := 1
a_2  while count < size do begin
         count := count + 1;
           fact := fact * count
         end
a_3  pop   b
62   pop   b
63   pop   b
```

```
count
fact
size

Symbol: size
  Item: var(4)
```

Figure 16 Interactive decompiling environmentshowing the symbol-table editing facility. See text for details.

The above arrangement can handle static variables and automatic variables (that is, those for which storage is allocated on the stack) from a single context (procedure or block). In general automatic variables must be be labelled with their context, for example `var(main, 2)`. This requires that the grammar-rules concerned be augmented with variables to record the context whenever a new stack-frame is established.

4.3 Recovering Data Types

In addition to recovering the control structures, the decompilation process should attempt to recover data type information. Simple (scalar) data types include those directly supported by machine instructions, for example in the case of the 8085, 8 bit bytes which may be interpreted as characters or integers and 16-bit words representing addresses or signed or unsigned integers, extended precision integers, floating point numbers of varying precision, ranges of integers, enumerated types (such as days of the week) and sets, typically represented as bit-maps. Complex data types include character strings, records (or structures) and arrays of the afforementioned simple and complex types. Modern languages, particularly those claiming to be object-oriented, allow the programmer to define a rich hierarchy of types (usually called classes) to represent concepts in the problem domain. We shall not consider such languages here; rather discussion will be confined to the data types provided for in languages such as Pascal.

While the problem of recovering type definitions is somewhat orthogonal to the use of DCGs, it is relevant to the question of the practicability of decompilation and we suggest here a general approach which involves, for scalar types, the following processes:

a. the recognition of the storage class of variables (number of bytes occupied, alignment) from the instructions used to access it.

b. assigning attributes to the type of the variable according to the operations which are performed on it (integer or floating point arithmetic, comparisons, bit-wise logical operations).

c. assigning variables to the same class where they are used in operations together and where attributes already assigned are compatible.

d. defining a class of variables as the transitive closure of the *same-class* relation defined by process c.

The blackboard model, referred to in section 4, provides a suitable framework for the application of such heuristics encoded as Prolog procedures. Attributes of variables can be asserted into the database as they are recognised. Examples of attribute assertions include:

```
size(var(1), 2).
size(var(2), 1).
participates_in(var(1), int_arith).
participates_in(var(2), byte_compare).
assigned_value(var(2), 1).
```

Recognised type compatibilities can be asserted thus:

```
same_type(var(1), var(3)).
```

These can be summarised as type declarations whenever a decompiled listing is requested.

In a strongly-typed language such as Pascal, integer subrange types might be recognised from run-time bounds-checks while enumerated types might be inferred from the set of constants assigned to variables of the type. Samples of values assigned to variables obtained from the run-time environment or data files would be of considerable benefit in determining the range and type of data, although relating external representations to internal values would require a deep understanding of input/output procedures.

In the absence of knowledge of the application domain, it is not possible to differentiate an integer subrange type [1 .. 7] from an enumerated type [sun, mon, tue, wed, thu, fri, sat] which may have been used by the original programmer. Such semantics may in some cases be provided as hints in an application domain knowledge base (for example, in an application dealing with dates, enumerated types *days of the week* and *months of the year* might be expected, as well as subrange types [1 .. 31] for *days of the month*, etc.). However, as with the assigning meaningful variable names, significant interaction with the analyst will be required. Assertions of correspondence between numeric and symbolic values for enumerated types might take the form:

```
symbolic_value(var(2), 1, monday).
```

The comments in the previous section regarding automatic variables from multiple contexts apply equally the recording of facts relating to type as they do do symbolic names. In addition it is desirable to minimise the scope of static variables in reconstructing declarations, even though program semantics may be the same with variables having global scope

5. CONSTRUCTING DECOMPILING GRAMMARS

The grammars for decompiling 8085 assembly code presented here have been discovered by examining the code generated by the Small-C compiler from known fragments of source code. In addition, run-time procedures were identified with the help of mnemonic labels. In fact these latter were generally quite short and their functions easy to determine. Automating the semantic analysis of such simple run-time procedures should not be difficult.

There will be some cases in practice where the compiler used to generate the program of interest will be known or can be guessed at. Software for embedded microprocessors is often compiled on development systems using languages provided by the chip manufacturer or by a major software vendor. Application programs on general purpose computers may be written using the standard system programming tools (for example, C under UNIX).

Further experimentation will be needed to determine the difficulty of constructing a decompiler when the source language and compiler are unknown and the extent to which this process might be automated. An even more difficult question is whether it is practicable to decompile to a high-level language, code which was actually written in assembly language or which has undergone intensive optimisation. It does seem likely that a language such as C, with its many constructs directly reflecting machine operations, would be a more promising target in this case than, say, Pascal.

6. DISCUSSION

This paper has described techniques for the construction of decompilers using definite clause grammars compiled or interpreted as Prolog programs. The principles have been illustrated using subsets of C compiled for the 8085 microprocessor. These experiments demonstrate that the use of DCGs and Prolog is a viable approach. In particular, the use of the Prolog database to store the initial assembly (or machine) code and recognised syntactic structures in the manner of a chart-parser supports an interactive approach and the application of additional knowledge. While the work described has not been carried through to the completion of a decompiler for a complete real-world programming language, Brushe [1990] describes a Prolog decompiler for a significant subset of PLM-80.

Avenues for further research include studies of different combinations of languages, compilers and target hardware, methods for handling compiler optimisation, the possibility of recognising high-level constructs such as loops and if-then-else statements in hand-written assembly code, and the use of heuristic knowledge of both the programming process and the application domain. Efficient implementation may be an issue for larger problems.

Finally we note that other approaches to the rapid construction of reverse engineering tools apart from the use of Prolog are possible. Other symbol manipulation languages such as Lisp could be chosen as a starting point, but generally would require significantly more work by the system builder. Kotik and Markosian [1989] describe the application of the REFINE programming language and environment to software re-engineering problems. REFINE provides tools for the construction of parsers and pretty-printers from context-free grammars, and a rule-based programming style for semantic processing of the resulting abstract syntax tree, however its purchase cost is many times that of the Prolog systems used in the current work, it demands a powerful workstation with a copious supply of memory and is less portable than Prolog.

REFERENCES

Aho, A.V. and Ulman, J.D. *Principles of Compiler Design*, Addison-Wesley: Reading, Massachusetts, 1977.

Alshawi, H., Moore, R.C., Moran, D.B. and Pulman, S. G. "The SRI Core Language Engine", Unpublished Report, SRI International, Cambridge, UK, 1988.

Bachman, C. "A CASE for Reverse Engineering", Datamation, 1 July, pp 149-56, 1988.

Bourne, S.R. *The UNIX System*, Addison-Wesley: Reading, Massachusetts, 1983.

Brushe, G. D. "Towards a Prolog De-compiler for De-compiling 8080 Assember Code into PL/M-80", Information Technology Divisional Paper, ITD-90-06, 1990.

CE Software *QuickMail Reference Manual*, 2nd Edition, CE Software Inc., Iowa, 1989.

Cohen, J. and Hickey, T.J "Parsing and Compiling Using Prolog", ACM Transactions on Programming Languages and Systems, 9, pp 125-163, 1987.

Colmerauer, A. "Les Grammaires de Metamorphose", Groupe d'Intelligence Artificielle, Université de Marseille-Luminy, Nov 1975. (Appears as "Metaorphosis Grammars" in: L. Bolc, Ed., *Natural Language Communication with Computers*, Springer-Verlag, Berlin, 1978.)

Horspool, R.N and Marovac, N. "An Approach to the Problem of Detranslation of Computer Programs", Computer Journal, 23, no 3, pp 223-229, 1980.

Intel *MCS 85 User's Manual*, Intel Corporation, California, 1977.

Jensen, K and Wirth, N. *Pascal User Manual and Report*, 2nd Edition, Springer-Verlag: Berlin, 1975.

Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*, Prentice-Hall: New Jersey, 1978.

Kotik, G. B. and Markosian, L. Z. "Program Transformation: The Key to Automating Software Maintenance and Re-engineering", unpublished report, Reasoning Systems, PAlo Alto, California, 1989.

Nii, H.P. "The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures", AI Magazine, Summer, pp38-53, 1986A.

Nii, H.P.	"Blackboard Systems from a Knowledge Engineering Perspective", AI Magazine, Fall, pp82-186, 1986B.
Pereira, F.C.N and Warren, D.H.D.	"Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks", Artificial Intelligence, 13, pp 231-278, 1980.
Spafford, E.H.	"The Internet Worm Program: An Analysis", Purdue Technical Report CSD-TR-823, Purdue University, Nov 1988.
Sterling, L. and Shapiro, E.	*The Art of Prolog*, MIT Press, Cambridge Massachusetts, 1986.
Warren, D. H. D.	"Logic Programming and Compiler Writing", Software-Practice and Experience, 10, Number II, pp 97-125, 1980.
Winograd, T.	*Language as a Cognitive Process, Volume 1: Syntax*, Addison-Wesley, Reading Massachusetts, 1983.

DISTRIBUTION

	Copy No.
Australian Defence Force	
Assistant Chief of the Defence Force (Development)	1
Commandant, Engineering Development Establishment	2
Director General, Engineering - Air Force	3
Commanding Officer, Electronic Warfare Operational Support Unit	4
Director General Naval Engineering Services	5
Defence Intelligence Organisation	
Scientific Adviser	6
Defence Signals Directorate	
Scientific Adviser	7
Defence Science and Technology Organisation	
Chief Defence Scientist)	
Central Office Executive)	8
Counsellor, Defence Science, London	Cnt Sht
Counsellor, Defence Science, Washington	Cnt Sht
Scientific Adviser, Defence Central	9
Naval Scientific Adviser	10
Air Force Scientific Adviser,	11
Scientific Adviser, Army	12
Director, Aeronautical Research Laboratory	13
Director, Surveillance Research Laboratory	14
Director, Materials Research Laboratory	15
Electronics Research Laboratory	
Director, Electronic Research Laboratory	16
Chief, Information Technology Division	17
Chief, Communications Division	18
Chief, Electronic Warfare Division	19
Research Leader, Information Processing and Fusion	20
Head, Software Engineering Group	21
Head, Trusted Computer Systems Group	22
Head, Command Support Systems Group	23
Head, Information Systems Development Team	24
Head, Image Information Group	25
Principal Research Scientist, Architectures	26
Principal Engineer, VLSI Design	27
Principal Research Scientist, C3I	28
Dr P. Dart	29
Mr R. Vernik	30

Dr S. Crawley	31
Dr. K. Mason	32
Author	33-36
Publicity and Component Support Officer, ITD	37-38
Libraries and Information Services	
Director of Publications, Australian Government Publishing Service	39
Technical Reports Centre, Defence Central Library, Campbell Park	40
Manager, Documentation Exchange Centre Defence Information Services	41
Defense Technical Information Center, United States	42-43
Defence Research Information Centre, United Kingdom	44-45
Director Scientific Information Services, Canada	46
Ministry of Defence, New Zealand	47
National Library of Australia	48
Defence Science and Technology Organisation Salisbury, Main Library	49-50
Materials Research Laboratory Library	51-52
Aeronautical Research Laboratory Library	53-54
Librarian Defence Signals Directorate, Melbourne	55
British Library Document Supply Centre	56
Spares	
Main Library, Defence Science and Technology Organisation Salisbury	57-76

Department of Defence

DOCUMENT CONTROL DATA SHEET

Page Classification	UNCLASSIFIED
Privacy Marking/Caveat	N/A

1a. AR Number AR-006-765	1b. Establishment Number ERL-0571-RR	2. Document Date SEPTEMBER 1991	3. Task Number DST 91/410

4. Title

DECOMPILING WITH DEFINITIVE CLAUSE GRAMMERS

5. Security Classification

Document	Title	Abstract
U	U	U

S (Secret) C (Confi) R (Rest) U (Unclass)

* For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L)

6. No. of Pages	28
7 No. of Refs.	20

8. Author(s)

S. T. Hood

9. Downgrading/Delimiting Instructions

N/A

10a. Corporate Author and Address

Electronics Research Laboratory
PO Box 1600
SALISBURY SA 5108

10b. Task Sponsor

11. Officer/Position responsible for

Security ....N/A............... ....

Downgrading....... . .N/A......................... ....

Approval for Release...........N/A...... ....

12. Secondary Release of this Document

APPROVED FOR PUBLIC RELEASE

Any enquiries outside stated limitations should be referred through DSTIC, Defence Information Services, Department of Defence, Anzac Park West, Canberra, ACT 2600.

13a. Deliberate Announcement

NO LIMITATION

13b Casual Announcement (for citation in other documents)

[✓] No Limitation

[ ] Ref by Author & Doc No only

14. DEFTEST Descriptors

De-compilers
Computer programs
Prolog (programming languages)

15. DISCAT Subject Codes

1205

16. Abstract

Decompiling is the process of deriving a computer program in a high-level language from one in machine-code or assembly language. Defence applications of decompiling include maintenance of obsolescent equipment, production of scientific and technical intelligence and assessment of systems for hazards to safety or security. This paper describes an approach to the rapid generation of decompilers through the use of Definite Clause Grammars., a class of abstract grammars which can be executed as Prolog programs. The approach is illustrated using "toy" languages. An environment which permits the integration of diverse sources of knowledge relevant to the decompilation problem and provides a graphical interface is described.

Page Classification	UNCLASSIFIED
Privacy Marking/Caveat	N/A

16. Abstract (CONT.)

17. Imprint

Electronics Research Laboratory
PO Box 1600
SALISBURY SA 5108

18. Document Series and Number	19. Cost Code	20. Type of Report and Period Covered
ERL-0571-RR	823350	ERL RESEARCH REPORT

21. Computer Programs Used

Prolog

22. Establishment File Reference(s)

N8306/2/1

23. Additional information (If required)