RETARGETABLE CODE GENERATION AND OPTIMIZATION USING ATTRIBUTE GRAMMARS

by

Mahadevan Ganapathi

Computer Sciences Technical Report #406

December 1980

RETARGETABLE CODE GENERATION AND OPTIMIZATION USING ATTRIBUTE GRAMMARS

BY

MAHADEVAN GANAPATHI

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1980

Retargetable Code Generation and Optimization using Attribute Grammars

Mahadevan Ganapathi

Under the supervision of
Associate Professor Charles Nicholas Fischer

Abstract

Attribute grammars are used to specify translations from an intermediate representation (a linear representation of parse-trees) to a target code representation of programs. A code generator may be obtained automatically for any compiler using attributed parsing techniques. A compiler built on this model can automatically perform most popular machine-dependent optimizations, including peephole optimizations. The code generator is also easily retargetable to different machine architectures. Implementations of a code generator based on this model exist for the VAX-11/780 and the PDP-11/70.

I dedicate this dissertation to

THE FLUTE PLAYER of the Jamuna banks

and also to

Koma Patti, my parents (Mahadevan and Sulochana), Uma, Kamala Babu, Bhanubhai and Indira mami

who have always provided encouragement and have been primarily responsible for my success in life

Acknowledgements

Charlie Fischer taught me parsing theory and compiler construction. He provided motivation, encouragement and transformed my ideas into a dissertation.

Raphael Finkel, Will Leland, Dave DeWitt and Bob Cook improved the readability of this thesis. Johannes Heigert, Steve Scalpone, Don Neuhengen and Keith Thompson provided invaluable help in the implementation of this thesis.

Ed Desautels and my colleagues at the Systems Lab have made my four years of stay very pleasant and most enjoyable. Some of them have been a source of many thought-provoking discussions (and also entertainment, especially during late hours!).

Don Dietmeyer has been a source of inspiration in logic design and digital hardware. Len Uhr, Larry Landweber and Jim Goodman have been a source of many pleasant conversations.

Finally, the Kohler Foundation and the Graduate School provided fellowship support for the past year.

I am grateful to all these people and I thank them all.

Contents

Chapter 1: Introduction

1.1 Motivation

The past decade has seen a number of attempts at automating the process of building code generators for compilers. Interest in this area is motivated by the following factors:

(1) Advances in hardware technology (microprogramming and VLSI) have led to the design and manufacture of a large number of diverse computer architectures (Intel-8Ø86, Z-8ØØØ, MC-68ØØØ, TMS-99ØØ).

(2) Advances in programming language design have led to the design of a large number of sophisticated programming languages (Pascal, Ada, Bliss, C, Fortran 77).

(3) Portable compilers producing high quality code for a variety of machine architectures are needed [Wulf 79].

(4) Such compilers must rest on a formalization of machine-dependent aspects of compilation including,

   (a) storage and temporary allocation,

   (b) code generation,

   (c) machine-dependent optimization.

1.2 Goals

The aims of this research are:

(1) to design an intermediate representation that is flexible enough to accommodate the diversity of popular programming languages,

(2) to use Attribute Grammars [Knuth 68] as a formal means of describing a machine architecture for purposes of code generation,

(3) to derive efficient code generators from such formal specifications and

(4) to generate high quality code by incorporating a large number of popular machine-dependent and peephole optimizations.

1.3 Code Generation Research

Previous research in code generation can be broadly classified into three categories: formal treatments [Newcomer 75, Aho 76], interpretive approaches [Elson 70, Richards 71, Wilcox 71, Donegan 73, Young 74, Ammann 77, Donegan 79] and descriptive approaches [Miller 71, Weingart 73, Snyder 75, Johnson 77, Fraser 77, Glanville 77, Ripken 77, Johnson 78, Glanville 78, Cattell 78, Cattell 79, Cattell 80, Graham 80, Wulf 80b]. For an extensive review and critique of these approaches, the reader may refer to [Ganapathi 80].

Formal treatments thus far have considered arithmetic expressions only. Interpretive approaches are improvements over ad-hoc code generation (because only P+M translators are needed to implement P languages on M architectures). But in such schemes machine descriptions are intermixed with the code generation algorithm. Retargeting thus requires changing the code generator for every new machine.

Descriptive approaches separate the machine description from the code generation algorithm, providing a higher degree of portability. In such schemes, pattern matching is used to replace interpretation. Fraser, Glanville, Ripken and Cattell have tried to derive code generators automatically from a machine description, although their methods are very different. They differ principally in

(a) the amount of processing that must be done by other parts of a compiler,

(b) the way multiple matches between machine instructions and source-language operations are handled,

(c) code generation speed and

(d) quality of generated code.

Fraser's rule based system is inefficient and its portability is questionable. He uses ad-hoc, machine-specific rules to perform storage allocation. His code generator is also very slow: It generates one line of assembler code each second on a PDP-1Ø KA1Ø. Furthermore, redundant loads and stores are often emitted. It is not clear how his code generator can use special machine instructions to yield optimized code (e.g. an increment instead of an add by one).

Ripken extended Aho and Johnson's algorithm [Aho 76] to generate locally optimal code. He also considered the interaction between different phases in a compiler. However, an implementation of his dynamic programming algorithm can be expected to be prohibitively slow.

Glanville's code-generation algorithm is derived from context-free parsing theory [Aho 73]. Storage is assumed to be bound by other phases of a compiler. His implementation is very efficient because standard context-free parsing techniques (which forbid back-up) are used. But, because of pure context-free matching, in certain cases it fails to generate optimized code (e.g. using indexing to avoid explicit addition in an addressing context). Furthermore, side effects and condition-code settings are ignored. Multiple matches of instruction patterns with the intermediate representation result in shift/reduce or reduce/reduce conflicts during bottom-up parsing. Such matches are always resolved in favor of the longest pattern. In many cases such a heuristic resolution fails to produce optimized code. For example, the VAX-11/780 has both two-address and three-address addition. Sometimes, a two-address op-code is preferable (e.g. for A := A + B), whereas in other instances a three-address op-code is better (e.g. for A := B + C).

Cattell designed a more complete code generator with a fixed set of operations for TCOL (an intermediate language). A recursive goal-directed heuristic search algorithm is used to derive code sequences in cases of operator mismatches between TCOL representations and target-machine templates. Such automatic derivation using axioms and a goal-directed heuristic search is not practical for a variety of machine instructions. Sometimes, it is very hard and time consuming (if not impossible) for a code generator to derive certain code sequences automatically.

These code sequences include floating-point operations on machines that do not support floating-point arithmetic, 2n-bit arithmetic on n-bit architectures and 'branch if equal' or 'branch if greater or equal' on the Intel-8Ø8Ø (which require a large number of instructions). Optimal code sequences that use auto-increment/decrement and recognize equivalent storage locations cannot be produced.

Glanville's scheme seems to be the most practical. However, his scheme requires other phases of the compiler to perform a significant amount of machine-dependent work (storage assignment, allocation of temporary operands, stack management, run-time display set up and linkage during procedure calls).

Our approach can be viewed as an extension to Glanville's scheme. Semantics and context in the form of attributes are used to control parsing of the intermediate form. Multiple matches between instruction patterns and the intermediate representation are resolved using disambiguating predicates [Milton 77]. The use of such predicates is a key point in this dissertation. Attribute grammar productions and disambiguating predicates not only provide considerable flexibility in retargeting the code generator, but also enhance the readability of the implementation and facilitate efficient optimizations.

We use a more complete machine description (including addressing modes and machine data-types, e.g. bytes, words, quadwords). Unlike Fraser or Cattell, we do not use ISP [Bell 71] (a CHDL: computer hardware description language) as a starting point. Lengthy code would be necessary to describe hardware stacks and floating point instructions. CHDLs are essentially programming languages with special features to describe digital hardware. Processing them to extract higher level abstractions (as needed in code generation) is as hard as writing a compiler for a programming language. For code generation it seems better not to spend a significant amount of effort translating procedural machine descriptions. Furthermore, in practice, an implementor using a machine manual can easily write new instruction descriptions [Graham 80]. Therefore, we resort to formal mechanisms for which easy and efficient translators are feasible. In light of the above, we re-state our research goals:

(1) structure the code generation process so that target machine dependency does not taint other phases of a compiler. Interfacing the code-generator package with other phases of a compiler should therefore become considerably easier.

(2) devise a simple and clean model of code generation and machine-dependent optimization using attribute grammars; ideally, one that is simpler and cleaner than Newcomer's and Cattell's means-end-analysis model [Newell 69].

(3) retain the speed and efficiency of Glanville's approach by using a fundamentally single-pass code generation scheme.

(4) include machine-dependent optimizations that have not been included in other portable code-generators. These optimizations include choosing between three-address and two-address instructions, subsuming (via auto-increment) additions widely separated from the current instruction (in effect, 'floating' an addition across many instructions), subsuming subtractions via auto-decrement in a similar fashion, removing redundant loads and stores, replacing memory references by register references, delaying operand movement into costlier storage locations and span-dependent optimizations [Szymanski 78, Szymanski 8Ø]. Such optimizations are hard to incorporate as a separate pass of peephole optimization [Fraser 79] since the instruction could have effects outside the window (e.g. condition-code setting and register contents).

1.4 Thesis Organization

The task of implementing a portable code generator is divided into:

(1) design of an intermediate representation (Chapter 2),

(2) attribute grammar machine description (Chapter 3),

(3) code selection (Chapter 4),

(4) machine-dependent optimization (Chapter 5).

Each chapter includes a brief review of the state of the art in its area. Implementation results, both for the VAX-11/78Ø and the PDP-11/7Ø, are presented in Chapter 6. Ideas for improvements to our implementation and future research in code generation are in the conclusions (Chapter 7). Details of the design of the intermediate representation and complete attribute grammar descriptions for the VAX-11/78Ø and the PDP-11/7Ø are included as appendices.

Chapter 2: Intermediate Representation

This chapter discusses the machine-independent phases of a compiler and the intermediate representation (IR) we will employ. Design considerations for a machine-independent/language-independent IR, which forms the input language for a portable code generator, are outlined. A Polish-prefix representation is used in our implementation. The use of attributes in such a representation to assign storage and set up displays is described.

2.1 Machine-Independent Phases

The major phases of a compiler's machine-independent structure are:

(1) lexical analysis (scanning),

(2) syntax analysis (parsing) with associated (optional) syntax error correction and recovery [Graham 79, Fischer 80],

(3) semantic analysis (e.g. type checking, procedure parameter checking and type coercions) and

(4) machine-independent optimizations, some of which are done at the source level while others are achieved as IR to IR transformations [Standish 76].

Examples of these machine-independent optimizations are:

(a) global flow analysis [Kennedy 71, Allen 72, Kildall 73, Ullman 75],

(b) optimizations involving constants, such as folding (excluding pointer arithmetic in address computations) and constant propagation [Aho 77], arithmetic on constants, strength reduction, replacing exponentiation to a constant power with multiplication and unrolling loops with constant step and limits,

(c) expression optimizations, such as common sub-expression elimination, expression transformations exploiting associative, commutative and distributive laws, arithmetic identities (e.g. addition to zero and multiplication by one), Boolean short-circuit evaluation and use of unary complement operators [Frailey 7Ø],

(d) pre-planning strategies that estimate optimum use of registers (e.g. global register allocation [Johnsson 75] and temporary allocation [Sethi 7Ø]),

(e) frequency reduction (moving operations out of frequently executed regions), such as hoisting loop-invariant computations and removing multiplication from a loop by detecting linear expressions involving induction variables (variables that assume values in an arithmetic progression) [Aho 77].

Not all of these optimizations are used in production compilers. Some are difficult to implement (e.g. detecting all induction variables in a program is unsolvable; optimal expression-evaluation order in the presence of common subexpressions is NP-complete); others may be unsafe (use of associativity with floating point addition) or irrelevant (Sethi-Ullman register optimization for stack computers or machines with asymmetric registers). However, portable code generators can use information gathered by some of these optimization algorithms (e.g. Johnsson's TNBIND and Sethi-Ullman computations) to bind registers. The IR should therefore provide a facility for allowing optimizing front-ends to express their resource allocation intentions. However, it must be the decision of the code generator to either satisfy these requests or ignore them, depending on the resources available in the target machine.

2.2 Design Considerations

Within a single compiler, an IR such as quadruples, triples or trees [Aho 77] is normally used for object code optimization. Among portable compilers an IR also serves to distinguish language-dependent issues from machine-dependent issues. Therefore, the design of an IR is critical to compiler portability, code generator portability and efficiency. Considerations involving the design of a common IR for a family of retargetable compilers are:

(1) The 'level' of an IR determines the work to be redone in either transporting a compiler to a new machine or using the same code generator for a new source language. If the level is too high, language dependencies creep in. Similarly if the level is too low, machine dependencies seem unavoidable.

(2) From UNCOL [Strong 58, Steel 61] experience, it seems impossible for a single IR to satisfy the requirements of all programming languages. To avoid compromises or inefficiencies, IRs should be flexible. Such a need is addressed in the Janus [Coleman 73] family of abstract machines and the TCOL [Wulf 80a] family of IRs. While Janus was developed to study portable compilers, TCOL is more ambitious in using the same family of IRs (e.g. $TCOL_{Ada}$, $TCOL_{Jovial}$) for, among other things, generation of verification conditions, language-oriented editing and code generation.

(3) The IR strongly influences the efficiency of code generation algorithms. Portable code generators commonly match the IR with instruction patterns of the target machine. Pattern matching in a tree is not as well understood as string matching. Most tree-matching algorithms rely on heuristics, and the results are not

provably correct. String matching is quite well understood, and efficient (i.e. linear) algorithms exist to parse strings. Also, typical parsers such as YACC [Johnson 75] use very simple drivers.

We will use a linearized Polish-prefix notation augmented with attributes (e.g. type, scope of a variable, register preference and context in which local evaluation takes place) to convey information to the code generator. The next section is a brief excursion into attribute grammars.

2.3 Attribute Grammars

Attribute grammars were proposed by Knuth [Knuth 68] as a means of formally specifying semantics within the context free grammar of a language (CFG). For a formal definition of attribute grammars or affix grammars, see [Koster 74, Lewis 76, Milton 77, Watt 77, Raiha 80]. Intuitively, each grammar symbol in a CFG is allowed to have a fixed number of associated values, termed attributes, whose domains may be finite or infinite. As an input is parsed, attributes are evaluated. The resulting syntax tree augmented with attributes represents the semantics of the input. The attributes associated with a given symbol may be synthetic or inherited. Synthetic attributes (which we denote by prefixing them with a '↑') are used to pass information up a parse tree. Inherited attributes (prefixed with a '$\downarrow$') are used to pass information down a parse tree. Each context-free production has an associated set of attribute evaluation rules. All rules can be packaged into predicate symbols (which check for attribute correctness) or action symbols (which compute new attribute values). To evaluate an action symbol, its inherited attributes are first made available. The action symbol is then applied; its synthetic attributes are computed as a result. This model makes it very easy to implement action symbols as subroutine calls.

Attribute grammars have been used as an effective tool to structure the translation phase of compilers [Watt 74, Lewis 76, Milton 79]. Our use of attribute grammars in code synthesis (storage allocation, code generation and machine-dependent optimization) should complement their traditional usage.

2.4 Attributed Prefix Notation

Linear notations can be classified as tuples (e.g. triples or quadruples that require explicit temporary specification) and strings (e.g. Polish-prefix, infix and postfix) that do not need explicit temporary specification. The number of temporaries required in the evaluation of an expression is often a machine-dependent issue. For example, the statement "A := B + C" requires one temporary on two-address machines (such as the PDP-11/7Ø) but none on three-address machines (such as the VAX-11/78Ø). The quadruple representation of the above statement is:

$(+, B, C, T_a)$

$(:=, A, T_a)$.

To generate three-address code requires a separate optimization pass to eliminate extraneous temporaries (T_a in the above example).

Infix notation is ambiguous without parentheses. Prefix notation is preferred to postfix for the following reasons (the discussion assumes a single-pass processing of the IR):

(1) Many architectures have non-orthogonal instruction sets. Some op-codes require operands to be in special machine locations (e.g. even-odd register pairs for multiplication and division on the IBM-37Ø and PDP-11/7Ø, registers for operand movement on the Intel-8Ø86). In postfix notation, an operand is encountered before its operator, and until the operator is seen the other associated operand is not known. The code generator might therefore have to back up in special cases to fix the operands (i.e. move them to valid locations).

(2) The context in which an expression is to be evaluated should be known prior to evaluation of the expression. For example, A < B requires an explicit Boolean result when evaluated in the context: C := A < B but does not need one in the context: IF A < B.

(3) Even if the IR-context does not require an explicit Boolean result, the instruction set of the target machine may nevertheless require creation of an explicit result. For example, a Boolean OR on the PDP-11/7Ø or the VAX-11/78Ø (bis) will always produce an explicit result. Furthermore, the VAX-11/78Ø provides a choice between a two-address (bis2) and a three-address (bis3) OR. In a conditional-statement context such as IF A OR B, bis3 will be used; in an assignment context (e.g. A := A OR B), bis2 may be preferable.

(4) Knowledge of the immediate destination of a result can influence the choice of operands for machine instructions. For example, a new temporary is unnecessary if the lefthand side of an assignment can store temporary results. It is therefore valuable to know the lefthand side of an assignment before encountering the righthand side.

The notation developed in this dissertation is essentially Polish-prefix with operators having a fixed number of operands. We augment this notation with attributes for operators and operands to carry forward semantic information essential for address-binding, code optimization and resolution of conflicts in cases of multiple matches during code selection. Some examples of attribute values we use are:

↑c	character data type
↑i	integer data type
↑l	long-integer data type
↑p	pointer data type
↑r	real data type
↑C	variable accessed through 'static chain'
↑D	variable accessed through display
↑E	external procedure or function
↑F	function or procedure name
↑G	global variable
↑L	local variable
↑O	push parameters in opposite (reverse) order
↑P	procedure or function parameter
↑R	register preference
↑S	variable to be placed in static area
↑T	optimize object-code for execution speed

We have a prototype IR processor that uses LEX [Lesk 79]. The complete IR specification is given in Appendix A. The following examples illustrate IR-form programs for binary search written in Modula and string comparison in C.

Notation:

:	beginning of declaration
{	opening of new scope
;	end of variable declarations
:=	assignment
@	indirection
#	address of variable
goto	unconditional branch
}	end of current scope
call	procedure or function call
relop	test and jump (takes two operands and a label)
Ørelop	relational with second operand implicitly Ø.

```
MODULE main;     (* Modula program to search for an item *)

USE      printd; (* system procedure for output *)
CONST    num_items = 10;
VAR      item     : ARRAY 1:num_items OF INTEGER;
         index    : INTEGER;
         result   : INTEGER;

         PROCEDURE printd (n:INTEGER);

         PROCEDURE binary_search (number:INTEGER):INTEGER;

         VAR      low, high, middle       : INTEGER;
         BEGIN
                  low     := 1;
                  high    := num_items;
                  REPEAT
                          middle := (low + high) / 2;
                          IF item[middle] >= number THEN
                                  low := middle + 1;
                          END;
                          IF item[middle] <= number THEN
                                  high := middle - 1;
                          END;
                  UNTIL low > high;
                  IF (low + 1) > high THEN
                          binary_search := middle;
                  ELSE
                          binary_search := 0;
                  END;
         END binary_search;

BEGIN    (* body of main program *)
         index := 1;
         REPEAT
                  item[index] := index;
                  INC (index);
         UNTIL index > num_items;
         result := binary_search (5);
         IF result <> 0 THEN
                  printd (result);
         END;
END main.
```

An experimental Modula front end produced the following IR:

```
: main  { " comments are enclosed within quotes "
: item ↑G ↑10 ↑1          " 10 times size of long datum "
: index ↑G ↑1 ↑1
: result ↑G ↑1 ↑1
: adritem ↑S ↑1 ↑p ↑R   " compiler-generated name "
;
: binary_search ↑1 ↑F ↑1          {" return long datum "
        : number ↑P ↑1 ↑1
        : low ↑L ↑1 ↑1
        : high ↑L ↑1 ↑1
        : middle ↑L ↑1 ↑1
        : arraybase ↑L ↑1 ↑p ↑R " Compiler-generated "
        ;
                := low  1
                := high  10
                := adritem  # item
L20     " repeat scope begins "
                := middle   /  + low high   2
        " increment adritem by middle * size of long "
                := arraybase + adritem  * middle SIZE↑1
        " if array element (a long datum) is less
          than number, branch to label  L21 "
                <  @↑1 arraybase  number  L21
                := low  + middle 1
L21             >  @↑1 arraybase  number  L19
                := high  - middle 1
L19             <=  low  high   L20 "repeat scope ends"
                <=  + low 1  high    L23
                := binary_search   middle
                goto   proced_end
L23             := binary_search  0
proced_end  }   " procedure declaration ends "
        := index 1
        " obtain address of first item "
        := adritem  +  # item  SIZE↑1
L30     " repeat scope begins "
        :=  @↑1 adritem  index
        " advance to next element of array "
        := adritem  + adritem SIZE↑1
        := index  + index 1
        <=  index  10   L30 " repeat scope ends "
        := result  call binary_search↑1  5
        0=  result   L31
        call  printd↑1 result
L31 }
```

The target code produced by our code generator for this IR is compared (in Chapter 6) with that produced by C compilers on the VAX-11/780 and the PDP-11/70 for the following equivalent C program:

```
#define NUM_ITEMS  10
int item[NUM_ITEMS+1];
int index;
int result;

binary_search(number)
int number;
{
        int low, high;
        register int middle;
        low = 1;
        high = NUM_ITEMS;
        do
        {
                middle = (low + high) / 2;
                if(item[middle] >= number)
                        low = middle + 1;
                if(item[middle] <= number)
                        high = middle - 1;
        }
        while(low <= high);
        if((low + 1) > high)
                return(middle);
        else
                return(0);
}

main()
{
        index = 1;
        do
        {
                item[index] = index;
                index++;
        }
        while(index <= NUM_ITEMS);
        result = binary_search(5);
        if(result != 0)
                printd(result);
}
```

As another example, consider the IR translation for the following C program:

```
#define SAME Ø
#define DIFF 1

main(argc, argv)
int argc;
register char **argv;

{       /* C program for string comparison */
        register char *arg;

        if(argc > 2) { /* more than 2 arguments */
                arg = *(argv + 1);
                /* check if argument '-p'
                 * appears on command line
                 */
                if(*arg == '-' && *(arg + 1) == 'p') {
                        /* compare first eight
                         * characters of next
                         * two arguments
                         */
                        if(strncmp(*(argv + 2),
                                   *(argv + 3),
                                   8)
                                        == SAME)
                                exit(1);
                }
        }
        exit(Ø);
}

strncmp(str1, str2, len)
register char *str1, *str2;
register int len;

{       /* string compare function compares character
         * by character; returns true if identical
         * otherwise returns false
         */
        register int i;

        for(i = Ø; i < len; i++)
                if(*str1++ != *str2++)
                        return(DIFF);
        return(SAME);
}
```

The hand-coded IR for the string-comparison program is:

```
: main ↑2 { " main is not a function; no ↑F is needed "
        : argc ↑P ↑l ↑1
        : argv ↑P ↑l ↑p ↑R
        : arg  ↑L ↑l ↑p ↑R
        ;
        <=  argc  2  L15
                := arg  @↑p + argv SIZE↑p
                <>  @↑c arg  45   "ascii -"   L2
                <>  @↑c + arg SIZE↑c  112 "ascii p"  L2
                Ø<>  call strncmp↑3
                                @↑p  + argv  * 2 SIZE↑p
                                @↑p  + argv  * 3 SIZE↑p
                                8
                                                 L3
                        call  exit↑l  l
L3
L2
L15
        call  exit↑l  Ø
}
: strncmp ↑3  ↑F ↑1     {
        : strl ↑P ↑l ↑p ↑R
        : str2 ↑P ↑l ↑p ↑R
        : len  ↑P ↑l ↑1 ↑R
        : i    ↑L ↑l ↑1 ↑R
        : temp ↑L ↑l ↑c ↑R
        ;
        := i Ø
        goto L25
L2ØØl
        := temp   =  @↑c strl  @↑c str2
        := strl  + strl SIZE↑c
        := str2  + str2 SIZE↑c
        Ø<> temp    L23
        := strncmp  l
        goto  L13
L23
        := i  + i  l
L25
        <  i len  L2ØØl
        := strncmp  Ø
L13
}
```

The code generated for this IR is given in Chapter 6.

2.5 Storage Assignment and Display Set-up

This section discusses some of the storage-assignment options offered by the current code-generator implementation. A compiler front end selects options by setting synthetic attributes of tokens.

At the IR level, variables are represented by their names, which are then bound to machine addresses before instruction selection. The decision of how to address locals and globals is not made at the IR level; it is treated as a code-generation issue. Some storage allocation and reclamation is done at well-defined times during execution (e.g. allocating space at block entry and releasing it at block exit). Other storage management is done at arbitrary moments (e.g. acquiring and releasing heap space). Storage assignment, that is, binding constants, simple variables and aggregates to machine storage-locations, may be based on a pre-planned strategy, such as global flow analysis, or done during code generation. In the former case, register preferences appear as attributes in the IR. For example, ': X ↑R' denotes that X is a variable that should preferably be placed in a register. Whether the code generator is able to satisfy requests for register assignment depends on the number of general-purpose registers in the target machine and the peculiarities of the instruction set (e.g.

even-odd register-pair use and ordering of operands in instructions).

Assembler instructions are used to allocate space for Global and Static variables (e.g. "a: .space 4" on the VAX-11/780). To support block structure, references to non-global variables are usually implemented through one of the following mechanisms:

(1) descending a 'static chain' of linked frames; elements in the chain are at a fixed offset from the frame pointer (FP) [Aho 77],

(2) displacement from the relevant display [Dijkstra 60]; the display is stored in a fixed location that is indexed off a display pointer (DP), usually a register,

(3) displacement from a display created at every procedure or block entry and placed on the stack [Gries 71]. In this case, the chain is descended only once per block (or procedure) entry instead of once per non-local reference.

The target architecture usually provides the FP (or a base register), DP and a hardware stack-pointer as registers. If they are not registers, then memory locations must be used to simulate them. Non-global variables whose space requirements are determinable at compile-time (e.g. integers, reals, characters and Booleans) can be bound to general-purpose registers, addresses with a fixed offset from the FP, addresses accessed indirectly through the DP or by explicit code sequences that descend a static chain that links frames. The current code-generator implementation supports all of these accessing methods.

The displacement from the frame may be positive or negative depending on the direction of frame growth; this information is provided as part of the machine description to the code generator. Variables that are both non-local and non-global are accessed through a display or through the static-chain mechanism outlined above. In these cases, another attribute (in the form of ↑number) specifies either the index of the relevant display or the number of levels of indirection from the FP (via the static link). This attribute is used to obtain the base register used to address the variable.

The machine data-type of an IR-variable is determined by searching a machine-description table, which provides information on data-types, their alignment restrictions and the maximum values storable in them. If the frame is part of the stack (as is usual in most architectures), the stack pointer may also have to be aligned. For example, the storage assignment and display set-up can be described as follows for the VAX-11/780:

```
#define MAXREG          15        /* max # general regs     */
#define USEREG          1Ø        /* max # scratch regs     */
#define DISPLAYREG      11        /* display pointer        */
#define ARGREG          12        /* arg pointer            */
#define FRAMEREG        13        /* frame pointer          */
#define STACKREG        14        /* stack pointer          */
#define STATICSTART     Ø         /* start of static area */
#define FRAMESTART      Ø         /* frame start offset     */
#define PARMSTART       4         /* parameters start off */
#define MAXALIGN        4         /* max align on stack     */
#define FRAMEDIRECTION  1         /* 1 = down, Ø = up       */
#define GSPACE          ".globl"
#define SPACE           ".space"
#define DSPACE          ".data"
#define TSPACE          ".text"
#define ARGALIAS        "r12 ap"
#define FRAMEALIAS      "r13 fp"
#define STACKALIAS      "r14 sp"
```

IR variables are thus converted into addresses before instructions are selected. In order to provide flexibility to allow any of the above block-structure mechanisms, our implementation of the code generator accepts multiple levels of both indirection and indexing (even indexing through a memory location). These machine-independent addressing modes are automatically mapped to the addressing modes of the machine by productions (details in Section 3.2). The usefulness of multiple levels of indexing is apparent on machines such as the Burroughs B-55ØØ, which needs no integer multiplication for array-element referencing. The subscript values are pushed on the stack, and the hardware uses a base descriptor and a subscript to obtain a descriptor for the correct row. Other subscripts are used to index the desired element. In our scheme, one could provide productions to capture this array indexing mode.

Addresses for dynamic arrays, strings and pointers are calculated and assigned at run time. Usually, for dynamic arrays, the dimensions of the array are known at block entry. Since space for arrays will be released at block exit, arrays can be assigned areas on the stack and accessed through a dope vector [Gries 71]. Dynamic strings and heap objects, however, cannot be stored on the stack. They need a heap and associated routines for heap management and garbage collection. Almost all machine architectures provide primitives for stack management. Comparable heap-management primitives are usually not available. Such allocation primitives are therefore normally realized as subroutine calls or as in-line code.

Procedure calls and returns require code (as procedure prologue and epilogue) to adjust the frame and stack pointers, save and restore displays and registers and pass arguments. Some architectures, such as the VAX-11/780, allow a variable number of registers to be saved and restored in a single instruction, thus lowering the overhead for a context-switch during a procedure call. In such a case, code has to be generated to specify which registers to save. Address assignment for procedure parameters can be relative to either the FP or a special argument pointer (e.g. AP on the VAX-11/780). The front end can indicate its choice of offset, whether positive or negative, from the FP or AP.

This choice may be overruled by the code generator if the machine architecture does not provide a facility for an implementation. For example, negative offsets from base registers are impossible on the IBM-370 and Univac-1100 series machines.

Another option provided in the current implementation is specification of the order of pushing procedure parameters on the stack (first argument pushed last or first). Parameter-pushing order affects the ability of target programs to interface with other system routines. Other implementation-dependent issues are the run-time startup routine (which is mostly language-dependent) and implementation of I/O calls (which is usually operating-system dependent).

In summary, IR variables are assigned storage before code selection. IR names are transformed to machine addresses, with machine-independent addressing modes that allow multiple levels of indirection and indexing. Consider the statement "A := B - 1", where A is a local variable and B is both non-local and non-global. The corresponding IR is:

```
    : {
            : A ↑L ↑1 ↑1
            " B is accessed through the static
              chain and is two levels outside "
            : B ↑C ↑2
            ;
            := A - B 1
    }
```

After storage binding:

```
    A becomes        , Disp↑a Base↑FP

    B becomes        , Disp↑b  @ @ , Disp↑s Base↑FP
                       (s is the offset of the static
                        chain from the frame pointer)
    1 becomes          Datum↑1.
```

The assignment statement becomes:

```
    := , Disp↑a Base↑FP
     - , Disp↑b @ @ , Disp↑s Base↑FP  Datum↑1.
```

The mapping of these addressing modes to actual modes provided by the target architecture is discussed in the next chapter. After parsing through addressing-mode productions (Section 3.2), the above statement becomes:

```
    := Address↑a - Address↑b Address↑c
```

where the attributes "a" and "b" represent the addresses of A and B, and the attribute "c" represents the constant 1. Chapter 4 describes the translation of this IR to target code using a single-pass algorithm.

Chapter 3: Attribute-Grammar Machine Description

Computer hardware description languages (CHDLs) have been traditionally used to describe, document and simulate complex digital systems. Register-transfer-level languages (AHPL [Hill 74], CDL [Chu 74], DDL [Dietmeyer 68, Dietmeyer 74, Dietmeyer 78]) describe digital hardware at the structural level. They are very useful during the initial stages of hardware design when the organization of the hardware and algorithms for implementing instructions are to be established. ISP [Bell 71] has been used to describe (informally) the instruction semantics for a large number of computers. It is intended to provide a behavioral description of the functioning of processors, viewed as programs. ISPL [Barbacci 76] is the first software-supported version of ISP. Its successor, ISPS [Barbacci 77], is implicitly oriented towards simulating the performance of an instruction set independently of the structural details of the hardware. To use ISPS as a starting point for software synthesis, ISPS descriptions have to be symbolically simulated by an interpreter. This simulation is a difficult task, as shown by Wick [75], who automatically generates assemblers, and Oakley [79], who automatically generates assertions. Furthermore, ISPS has some limitations: It is not suited for describing special machine instructions such as the CDC-Star vector operations or the IBM-360 translate-and-test instruction, or addressing modes such as the auto-increment/decrement on the PDP-11/70 and Motorola-68000. Lengthy ISPS code is necessary to describe hardware stacks, floating point and block-move operations.

In our scheme, the information necessary for IR-operation selection is described by attribute-grammar productions, with at least one template for every IR operator. They are described under the general categories of addressing-mode productions and instruction-selection productions.

3.1 Architecture Primitives

Code generation requires descriptions of the following components of a machine architecture:

(1) addressable units for storing source-language values (e.g. memory, registers and hardware stack),

(2) a run-time display mechanism such as display pointer, activation pointer (which may be a register or a memory location) and direction of frame growth (either up or down),

(3) the set of instructions available for implementing IR operations; their execution time and size,

(4) primitive data types (data objects having direct hardware realization) that can participate as operands

to instructions, the maximum value they can hold and their contribution to instruction size,

(5) addressing modes available to access and retrieve operands,

(6) side effects of instructions, such as condition-code setting and even-odd register pair usage,

(7) assembler or binary formats of instructions and addressing modes.

The hardware abstractions essential to code generation are data types, addressing modes and instructions. Data types are groups of bits that can participate as operands to instructions. Some examples are:

VAX-11/78Ø	byte, word, long, quad, float, double,
Z-8ØØØ	bit, byte, word, long, quad, BCD,
Intel-8Ø86	byte, word, BCD,
MC-68ØØØ	bit, byte, word, long, BCD.

The interpretation of these bits by the central processing unit depends on their representation (e.g. signed magnitude, two's complement).

Addressing modes are access paths to retrieve operands residing in storage locations such as memory, stack or register. The time taken to access an operand depends on the access path and the storage location in which the operand resides. For example, it is faster to retrieve an operand from a register than from memory or the stack. The timings may also be dependent on the presence of a cache, a floating-point accelerator, pipelining or other configuration details. The size of a machine instruction is also affected by the addressing mode. Addressing-mode productions describe patterns for address formation in the target architecture (details are in the next section). The assembler formats for these modes are tabulated. Appendix B contains addressing-mode tables for the VAX-11/780.

The operations of machines can broadly be classified, with respect to mapping IR operators to machine op-codes, under the following categories:

(1) Data-transfer instructions are used to implement source-language assignments to variables. Assignments can sometimes be subsumed as part of other operations (e.g. A := A + B can be implemented as add B, A). Assignments of aggregates may not be implementable in a single data-transfer instruction; often a series of moves or a loop is required to implement them.

(2) Arithmetic instructions are used to implement arithmetic-expression evaluation and address calculations.

(3) Boolean instructions are used to implement Boolean-expression evaluation under two contexts: (a) as values to be manipulated or assigned and (b) as predicates to control constructs.

(4) Control instructions (comparisons and branches) are used to implement relational operators, sometimes including an implicit comparison with zero. Often the result of a comparison is a condition-code setting that is subsequently tested to decide the control flow of the user-program.

(5) Subroutine call and return instructions are necessary to implement procedures.

(6) Special instructions are used to optimize object code. Examples include:

(a) single instructions that effectively perform combinations of arithmetic and control operations (e.g. 'subtract one and branch' on the PDP-11/70, 'add one and branch if less than' on the VAX-11/780), and

(b) shift instructions on integers (often used to replace integer-multiplication and, in some cases, division by a power of two).

Appendix C contains op-code description tables for the VAX-11/780.

3.2 Attribute-Grammar Productions

For purposes of pattern matching and instruction selection, the instruction set of the target architecture is represented as a set of attribute-grammar productions.

These productions form the input to a program that generates a code generator for the target machine. This section illustrates the use of attribute-grammar productions; the attributed parsing algorithm is discussed in Section 4.3.

All productions are of the form 'LHS → RHS', where LHS stands for lefthand side, RHS for righthand side. The LHS is a single non-terminal usually appearing with synthetic attributes. The RHS contains:

(1) terminals with synthetic attributes,

(2) non-terminals with synthetic attributes,

(3) disambiguating predicates (underlined) with inherited attributes and

(4) action symbols (capitalized) with synthetic and inherited attributes.

Attribute occurrences may be constants or variables. Constant attributes (with the exception of self-defining constants) are enclosed within quotes. An attribute variable is a shorthand referring to a data structure that contains all the attributes of some symbol. The same attribute variable may appear more than once in a production. In such cases attribute values are implicitly copied from synthetic attributes of a symbol in the RHS to synthetic attributes of the LHS or to inherited attributes of disambiguating predicates and action symbols. For example, in the production:

Byte$\uparrow$a $\rightarrow$ Address$\uparrow$a,

the attribute variable a is copied from symbol Address to Byte.

Disambiguating predicates do not compute new attribute values. They yield true or false only. The disambiguating predicates of each production are included to determine when the production is applicable (i.e., when it should be selected as a template for code generation), e.g.,

Byte$\uparrow$a $\rightarrow$ Address$\uparrow$a <u>IsByte</u> ($\downarrow$a)

This production is used only if Address has attributes that show it is a byte. A production is applicable only if all its disambiguating predicates evaluate to true. In order to guarantee that at most one production is selected, productions are tried in order of specification. In general, a hierarchy of disambiguating predicates can be designed to select only one production. The ordering could be selected for either decreasing object-code space or increasing execution speed. Our experience suggests that a single linear ordering usually suffices.

The kinds of productions needed for an entire code generator can be broadly classified into addressing-mode productions and instruction-selection productions. Although examples in this section pertain to the PDP-11/70 and the

VAX-11/78Ø, the technique is generally applicable and feasible, as demonstrated by our specific implementations.

<u>Addressing-mode productions</u>:

Each production has an RHS specifying the pattern of an IR addressing mode. The production creates the proper machine address (in an action symbol). For example, the following production is used to specify an index addressing mode on the PDP-11/7Ø or a displacement addressing mode on the VAX-11/78Ø (',' denotes indexing in the IR):

$$\text{Address}{\uparrow}a \rightarrow \text{ , Disp}{\uparrow}b \text{ Base}{\uparrow}c \quad \text{ADDR } ({\downarrow}b{\downarrow}c{\uparrow}a)$$

"Disp" represents a local variable with attributes specifying the machine data type and offset from a frame pointer. These attributes are determined when IR variables are bound to locations in the target machine. The attribute variable "c" specifies the base (or display) register of the IR variable. The action symbol ADDR synthesizes an address for a datum on the target machine. The attribute "a" represents this address; in our implementation, it has the following components:

(1) a base register,
(2) an offset from the base register,
(3) an optional level of indirection,
(4) an index register (if any) and
(5) the name of a variable (in case it is global).

These components may vary when the code generator is retargeted to new machines. However, for a variety of machines, including the VAX-11/78Ø, IBM-37Ø and the PDP-11/7Ø, this structure seems to suffice. The addressing mode productions determine the components used. For some machines, a component may never be necessary. For example, on the PDP-11/7Ø, the index-register field will never be used (on the PDP-11/7Ø, the index register and the base register cannot be used simultaneously). The addressing-mode productions reflect addressing modes supported by the target machine. If the target machine does not support simple addressing modes, code sequences may be needed for addressing purposes. For example, if a machine does not support indexing, the IR will be parsed by other productions that represent simpler addressing modes; code for composing those modes will be generated.

Addressing-mode productions are augmented by a few productions that map machine-independent addressing modes to target addresses (such as multiple levels of indirection and indexing, as discussed in Section 2.5). For example, if the target machine has no display registers, then a variable is indexed from a memory location. Most architectures require the index to be a register. The following productions force an index to be located in a register:

```
        Base↑a → Modes↑a IsReg (↓a)

        Base↑a → Modes↑b
                        GETREG (↓'long' ↑a)
                        EMIT (↓'movl' ↓b ↓a)
```

The non-terminal Modes represents any addressing mode supported by the target architecture. The predicate IsReg checks if "a" is already in a register. If not, a register "a" is obtained from the action symbol GETREG, and "b" is moved to "a". The action symbol EMIT takes a machine op-code, addresses, and labels (optional) as input and formats target code with the help of machine-description tables. Architectures usually support one level of indirection. The following productions implement one level of indirection on the PDP-11/70 (@ specifies indirection in the IR):

```
        Address↑a → IndirectModes↑a

        IndirectModes↑a → @ DirectModes↑b NotIndirect (↓b)
                                ADDR (↓@ ↓b ↑a)
```

The predicate NotIndirect ensures that "b" does not specify an indirect addressing mode (i.e. its indirection flag is not set). ADDR turns on the indirection flag for datum "b". However, the IR may have multiple levels of indirection (e.g. when descending a 'static chain' that links blocks). To implement more than a single level, two more productions are needed:

```
        IndirectModes↑b → AnotherLevel↑b

        AnotherLevel↑b → @ IndirectModes↑a
                               GETREG (↓'word' ↑r)
                               EMIT (↓'mov' ↓a ↓r)
                               ADDR (↓@ ↓r ↑b)
```

Similarly, multiple levels of indexing require a few extra productions. Thus, each IR variable is converted into an address (or datum) with an attribute that is automatically synthesized using productions. Some addressing modes, such as auto-increment, modify the address of the datum after usage. They are used to subsume addition or subtraction in the context of address calculations (Section 5.3).

<u>Instruction-selection productions</u>:

Each production has an RHS specifying the pattern in the IR and the corresponding code sequence to be emitted on a match. The LHS may be an explicit result location (a register or a memory location), in which case it specifies the data type of the result, or a condition code location, or simply a non-terminal place-holder. Consider addition on the VAX-11/780. There are two-address and three-address add op-codes. Furthermore, the increment instruction can be used for adding one. For a byte datum, these three forms of addition are expressed as follows:

```
Byte↑r  → + Byte↑a Byte↑r IsOne (↓a) IsTemp (↓r)
                EMIT (↓'incb' ↓r)

        → + Byte↑r Byte↑a IsOne (↓a) IsTemp (↓r)
                EMIT (↓'incb' ↓r)

        → + Byte↑a Byte↑r TwoOp (↓+↓a↓r)
                EMIT (↓'addb2' ↓a ↓r)

        → + Byte↑r Byte↑a TwoOp (↓+↓a↓r)
                EMIT (↓'addb2' ↓a ↓r)

        → + Byte↑a Byte↑b
                GETTEMP (↓'byte' ↑r)
                EMIT(↓'addb3'↓a↓b↓r)
```

The first and second productions specify the addition of 1 to "r". Both productions are needed to represent the commutativity of addition. In case either production is selected, the op-code incb (increment byte) is emitted. The non-terminal on the LHS (Byte) and its attribute (r) specify the data type and address of the result respectively. The third and fourth productions specify two-address addition of "a" and "r" using op-code addb2. Similarly, the last production specifies three-address addition of "a" and "b" using op-code addb3. In this case, the sum is stored in "r" that is obtained from action symbol GETTEMP. The location "r" may be a free register or the LHS of an assignment statement whose previous contents need not be preserved.

An addition of two IR data in byte format will match the RHS of one of these productions. The choice of the RHS is determined by attribute values and the disambiguating predicates. If an operand is 1 then an incb instruction is selected. Productions three through five handle addition of a constant other than 1. In an assignment context, a global attribute keeps track of the target address of the assignment statement. The disambiguating predicate TwoOp evaluates to true if either operand is the target of assignment or its value need not be preserved after addition. Consequently, a two-address addb2 is selected. If TwoOp evaluates to false, then a three-address addb3 is selected.

3.3 Transfer Code Sequences

Operands may be intentionally relocated by the code generator to storage locations other than the one in which they normally reside, for any one of the following reasons:

(1) Destructive operations:
Many machine operations, such as two-address instructions, destroy the contents of a participating operand. For example, on the PDP-11/7Ø, "add A, B" destroys the contents of location B. Thus, to implement "C := A + B", either B or A must be moved to a temporary location before addition, or a three-address instruction must be used.
(2) Data-type conversion:
Most machine op-codes operate only on operands of identical data types (except on tagged architectures [Feustal 73]). Mixed-mode operations are therefore implemented by converting all operands to the same machine data type before performing the operation. Thus, the statement C := B + C, where B is an integer and C a floating-point number, is implemented by converting B to a floating-point data type and then adding B to C. Such conversions are either specified by compiler front-ends as type coercions or are automatically performed by the code generator (e.g. when both B and C above are integers but B occupies a byte data-type and C occupies a word data-type). To implement data-type conver-

sions, some machines provide a special conversion instruction (e.g. 'cvtbw' on the VAX-11/78Ø) whereas other machines might require a sequence of instructions.

(3) Instruction set non-orthogonality:

The orthogonality of an instruction set is the regularity with which any op-code can be used with any machine-primitive data type and addressing mode. Every architecture designed and marketed so far possesses some amount of non-orthogonality. For example, on the Z-8ØØØ and Intel-8Ø86, no memory-to-memory arithmetic is possible. On the PDP-11/7Ø and IBM-37Ø no memory-to-memory multiplication or division is possible, but memory-to-memory addition and subtraction are allowed. Such irregularities force the code generator to produce extra code for relocating operands. To implement "C := B * C" on the PDP-11/7Ø, where both B and C are integers in memory locations, C has to be relocated to an even register of an even-odd pair. Consequently the corresponding odd register may need to be relocated before the multiplication so that its contents are not destroyed as a side-effect.

Transfer code sequences implement forced operand relocations. They are specified as part of the RHS of a transfer production. For example, to convert a word datum to a long datum on the VAX-11/78Ø, the following transfer production is used:

```
Long↑a → Word↑b ConvToLong (↓b)
                GETTEMP (↓'long' ↑a)
                EMIT (↓'cvtwl' ↓b ↓a)
```

If such transfer code-sequences are not provided, the code generator may block while parsing a semantically correct IR input. Usually, simple moves are adequate, but sometimes lengthy code sequences are necessary.

In summary, the components of target architectures needed for instruction selection are described as attribute-grammar productions to a generator for the target machine's code generator. The next chapter describes the translation of IR to target code using transition tables automatically produced by the code-generator generator.

Chapter 4: Code-Selection Issues

Code generation is the process of mapping some intermediate representation of the source program into assembly or binary machine-code. This complex task involves selecting machine instructions to implement programming language constructs for all of the following operations:

(1) storage assignment,

(2) accessing variables and selecting addressing modes,

(3) setting up run-time display linkage,

(4) procedure body prologue and epilogue,

(5) evaluating arithmetic and Boolean expressions,

(6) executing control constructs and evaluating Boolean expressions that do not need to store an explicit result.

The attribute grammar for the target machine is input to a code-generator generator (CGG) whose output is a specific code generator for the machine. The code generator consists of a set of transition tables and a driver for these tables. This driver serves as a push-down automaton that parses the IR form. Instructions (machine operations) are selected during parsing. To transport compilers to a new machine, the attribute-grammar description of that machine is given to the CGG. Transition tables for the machine are then automatically obtained and the same driver is used.

4.1 Code-Generator Generator

The CGG constructs a context-sensitive parser [Watt 74, Watt 77]. The parser constructor is a generalization of context-free parsing methods that accepts a useful class of attribute grammars: those that are amenable to single pass, left-to-right parsing. Apart from the evaluation of action symbols, resulting parsers from such constructors retain the linear performance characteristics of context-free parsers. Watt has related the parsing problem of attribute grammars to the context-free parsing problem. He has decomposed construction of attributed-grammar parsers into three stages:

(1) To guarantee correspondence between symbols on top of the parse stack of the code generator and their associated attributes on top of the attribute stack, copy symbols (special null non-terminals) are introduced into the grammar. A new grammar (called the head grammar) is automatically formed from an attribute grammar, with production rules stripped of their attributes and augmented with copy symbols. This construction is independent of the parsing method to be adopted and is detailed in [Watt 77].

(2) A context-free parser is constructed from the head grammar.

(3) The context-free parser is generalized to include an attribute stack to deterministically parse attribute grammars.

4.2 Instruction Pattern-Matching: Attributed Parsing with Contextual Predicates

Both top-down and bottom-up parsers have proved attractive to language implementors since they organize the translation phase of compilers. This section discusses the use of parsing techniques to organize a compiler's code-generation phase.

Top-down (LL) parsing is not well suited to matching prefix IRs against prefix target-machine templates. An operator in the IR (say +) corresponds to many templates beginning with the same operator (e.g. 'incb', 'inc', 'add' on the PDP-11/7Ø). In top-down parsing, production identification takes place before all of the RHS components have been processed. Ambiguities that occur in LL parsing are all 'predict-predict' conflicts. A disambiguating predicate (also called a contextual predicate [Milton 77]) can be associated directly with the production whose prediction it will determine. Since, before prediction, very little information is available on the operands, many lookaheads are required to select the proper template. Contextual predicates need to consider the non-terminal on top of the parser stack along with these look-aheads.

In contrast, bottom-up parsing is better suited to instruction selection because a reduction takes place only when the entire RHS of a production has been processed. All information on operands, available as attributes of symbols on the RHS, can therefore be used to disambiguate multiple matches and to control parsing. However, attribute grammars must be restricted in the following ways to make them suitable for attributed bottom-up processing.

(1) Since the proper actions to perform depend on identifying the associated production, action symbols may appear only at the extreme right end of a production. However, this restriction may be lifted in certain special cases where occurrences of action symbols before the right end of productions can be automatically replaced by non-terminals that generate the empty string (and thus serve as markers). If such movement of action symbols to the left is inappropriate, an unresolvable parsing conflict will arise [Watt 77].

(2) Bottom-up parsers operate by constructing forests of derivation sub-trees and then piecing them together. Information flowing down a sub-tree (in inherited attributes) cannot guide a parse, since by the time such information becomes available the entire sub-tree has already been con-

structed. Furthermore, information cannot flow from one sub-tree to a sibling sub-tree, since the fact that they are siblings is not established until both have been constructed. All attributes of non-terminals must therefore be synthetic.

The flow of contextual information is therefore highly restricted. In the absence of action symbols, information can only flow strictly up the tree, while an action symbol node can receive information only from siblings to its left.

Contextual predicates take a fundamentally different form for LR parsers than for LL parsers. In LR parsing the conflicts are of the 'shift-reduce' or 'reduce-reduce' variety. Moreover, the conflicts are present only in the context of a particular state or configuration set. Thus, while an LL parser bases its decision on a non-terminal and a look-ahead, an LR parser bases its decisions on a parse state and a look-ahead. Disambiguating predicates are therefore associated with states, not productions. Furthermore, the top stack symbol (along with its attributes) will typically not provide enough left context for a predicate to perform disambiguation, due to restriction (2) above. Left context in bottom-up parsing can only be transmitted up the derivation tree. Thus the symbol on top of an LR parser stack can only convey information (in its synthetic attributes) from the sub-tree it heads. Therefore, in general, disambiguating predicates will need to examine more than one of the symbols at top of the stack. The state (actually, the corresponding configuration set) will determine how many symbols on top of the stack will be available to the disambiguating predicate.

The disambiguating predicates of each production are usually written as the productions are designed. They serve as a guide to when the production is applicable. In most cases, these predicates also serve to resolve parsing conflicts (i.e. they control parsing). In practice, some disambiguating predicates are added only after a canonical collection of configuration sets has been computed and found to contain conflicts. Upon the occurrence of a parsing conflict, disambiguating predicates of successive productions in the current state are polled to determine the one whose attributes allow it to be applied (predicates are not ignored if there is no conflict). Disambiguating predicates are implemented within the standard framework of attribute evaluation by an evaluation rule that examines the attribute stack and checks for applicability of the productions.

In summary, one-pass disambiguated bottom-up attributed parsing requires that:

(1) Non-terminals must have only synthetic attributes.

(2) Inherited attributes of action symbols must depend only on the attributes of symbols to their left in the production.

(3) Each production includes optional disambiguating predicates to control production recognition.

4.3 Code-Generation Algorithm

The attributed bottom-up parser with disambiguating predicates employs the standard LR(k) parsing loop with added code to manipulate attributes. Although using two stacks (the control stack and the attribute stack) aids conceptual clarity, in practice all attributes of a given symbol can be packaged into a single data structure with a pointer to it (the attribute variable) kept on the control stack. Since the set of attributes is relatively small (the prototype code generator uses ten attributes in all, covering many architectures), the parser does not need to be able to handle fully general attribute sets.

In our notation, RHS symbols with constant attribute values differ significantly from RHS symbols with symbolic (variable) attributes. Symbols with constant attribute values can only match corresponding values in productions. Datum↑2 will only match a datum with an attribute value of 2, whereas Datum↑a can match a datum with any attribute value. In case the same attribute variable appears several times within one production, attribute values need not be copied; the relative offset (from the stack top) of the defining instance of the attribute variable is carried forward. On a shift operation, attribute values of a symbol are copied onto the stack. On a reduce operation, action symbols are processed and synthetic attributes are returned to the LHS symbol of the production. For each action symbol, in turn, its inherited attributes are first evaluated, and then the corresponding function (the action symbol) is called to evaluate its synthetic attributes. The algorithm is detailed below:

Notation:

Token	current IR token being scanned.
LaToken	look-ahead token.
$\uparrow$Sym	synthetic attributes of Sym.
Stack	control stack of IR parser.
Stack[d]	dth symbol from top of the stack (Stack[1] is the top).
Push(X)	push X onto the stack.
Pop(N)	pop N symbols off the stack.
Func(Inh, Syn)	action symbol with Inh inherited and Syn synthetic attributes.
Prod	production recognized.
Lhs_{Prod}	LHS of Prod.
Rhs_{Prod}	RHS of Prod.
State	current state in the code-generator automaton.
Nextstate(State, Symbol)	determines the next state of the automaton.
Nextaction(State, Symbol, $\uparrow$Symbol, LaToken)	determines the set of possible actions that could be performed when shifting Symbol (with its synthetic attributes) in state 'State' and the look-ahead LaToken.
Actset	set of possible actions determined by Nextaction.
Disambiguate(State, Actset)	predicate that takes the current state and action set as input and returns only one action as output. The attributes on the stack are available for disambiguating purposes.
Action	current operation of the driver.

```
PROGRAM CodeGenerator;

State    := Ø;
Action   := Shift;

    SWITCH (Action) OF

    CASE Shift:
        Push(State);
        (* stack the token's synthetic attributes *)
        Push(↑Token);
        (* determine next action *)
        Actset := Nextaction(State,Token,↑Token,LaToken);
        IF Actset is single-valued THEN Action := Actset
        ELSE Action  := Disambiguate(State, Actset);
        (* determine next state *)
        State := Nextstate(State, Token);
    END; (* case shift *)

    CASE Reduce:
        SWITCH ($Lhs_{Prod}$) OF

                CASE actionsymbol: (* Func(Inh, Syn) *)
                    Func(Stack[1],..,Stack[Inh],
                         Stack[Syn],..,Stack[1]);
                END; (* case action symbol *)

                CASE nonterminal:
                    Pop($Rhs_{Prod}$);
                    Pop(↑$Rhs_{Prod}$);
                    State := Stack[1]; (* top of stack *)
                    Push(↑$Lhs_{Prod}$);
                END; (* case nonterminal *)
        END; (* switch *)
        Actset :=
        Nextaction(State,$Lhs_{Prod}$,↑$Lhs_{Prod}$,LaToken);

        IF Actset is single-valued THEN Action := Actset
        ELSE Action  := Disambiguate(State, Actset);
        State    := Nextstate(State, $Lhs_{Prod}$);
    END;    (* case reduce *)

    CASE Accept:    halt, accepting;
    END;    (* case accept *)

    CASE Error:     halt, rejecting;
        (*the front-end generated an invalid IR sequence*)
    END;    (* case error *)
    END; (* switch *)
END. (* end program codegenerator *)
```

4.4 Examples of Parsing using Attributes

In this section we illustrate examples of using attributed parsing to generate VAX-11/780 code. These examples emphasize the PDP-11/70 and the VAX-11/780. However, attributed parsing is a generally applicable technique for compiler code generation and optimization usable on almost any architecture. Its feasibility is demonstrated by specific implementations.

Consider the translation of the statement "A := B - 1" on typical architectures with several lengths of integers (e.g. byte, word, long). The IR after storage-binding is:

:= Address↑a - Address↑b Address↑c

where the attribute variable a includes 'long' and address information for A, b has 'word' and other information for B, and c has 'byte' and 1 as the actual value. We now trace the parsing process.

(1) The following production is recognized:

Long↑a → Address↑a <u>IsLong</u> (↓a)

This production matches any Address with attributes that declare that its type is long. Because a appears twice in this production, it is implicitly copied from Address to Long. Thus, the attributes of A are carried forward. We now have

:= Long↑a - Address↑b Address↑c

(2) Next, production

$\text{Word}\uparrow a \rightarrow \text{Address}\uparrow a\ \underline{\text{IsWord}}\ (\downarrow a)$

matches Address↑b, because it is a word. The local attribute variable "a" is instantiated as "b". We reduce the IR further to:

$:= \text{Long}\uparrow a\ -\ \text{Word}\uparrow b\ \ \text{Address}\uparrow c$

(3) Now, the following production is matched:

$\text{Long}\uparrow a \rightarrow \text{Word}\uparrow b\ \underline{\text{ConvToLong}}\ (\downarrow b)\ \text{GETTEMP}\ (\downarrow\text{'long'}\uparrow a)$
$\text{EMIT}\ (\downarrow\text{'cvtwl'}\downarrow b\downarrow a)$

We convert from word to long format by first allocating a temporary (say register r_1) through the action symbol GETTEMP, then issuing a 'convert word to long' instruction through the action symbol EMIT. We now have reduced the IR to:

$:= \text{Long}\uparrow a\ -\ \text{Long}\uparrow r_1\ \ \text{Address}\uparrow c$

(4) Next, the constant 1 is reduced to a Long by the production

$\text{Long}\uparrow a \rightarrow \text{Address}\uparrow a\ \underline{\text{IsLong}}\ (\downarrow a)$

We have reduced the IR to:

$:= \text{Long}\uparrow a\ -\ \text{Long}\uparrow r_1\ \ \text{Long}\uparrow c$

(5) The following production is now matched:

$\text{Long}\uparrow b \rightarrow -\ \text{Long}\uparrow b\ \text{Long}\uparrow c\ \underline{\text{IsOne}}\ (\downarrow c)\ \text{EMIT}\ (\downarrow\text{'decl'}\ \downarrow b)$

This production describes a special-purpose decrement instruction, applicable only if the second operand is the constant 1. We have reduced the IR to:

$:= \text{Long}\uparrow a\ \text{Long}\uparrow r_1$

(6) Finally, the following production is matched:

```
Instruction → := Long↑a Long↑b
                        IF NotEquate (↓a↓b) THEN
                        DELAY (↓'movl' ↓b ↓a)
```

<u>NotEquate</u> evaluates to false if "a" and "b" are equivalent locations and consequently, reducing by this production does not produce any code. DELAY is a variant of EMIT that can delay generation of an instruction pending future instructions. In this case, the move of r_1 to A is delayed so that future references to A can be replaced by r_1. Also, the move may be completely suppressed if, for example, another assignment to A is encountered before it is referenced.

The use of attribute values to control parsing the IR allows us to significantly improve the quality of generated code with little effort. For example, in step (5), above, if the left operand had not been in a temporary, we would have generated two instructions (a 'move', then the decrement). A better code sequence would be to use the VAX's three address format to generate, for example, "subl3 1, B, r_1". To include this optimization we add a disambiguating predicate "IsTemp" to the Decrement production to obtain

```
Long↑a → - Long↑a Long↑b IsOne (↓b) IsTemp (↓a)
                        EMIT (↓'decl' ↓a)
```

If the a's attributes show it is not a temporary, IsTemp evaluates to false, and recognition of this production is blocked. Instead, an equivalent (but longer) instruction is generated by this alternate production:

```
Long↑a → - Long↑b Long↑c GETTEMP (↓'long'↑a)
                         EMIT (↓'subl3' ↓c ↓b ↓a)
```

Many machines, including the VAX, contain specialized hardware features that are difficult to exploit in compiler-generated code. A good example is the auto-increment/decrement feature. Compilers find it difficult to recognize the special cases in which a subtract operation can be subsumed in a later instruction (or an add operation in an earlier instruction) by auto-decrement (auto-increment). Our approach can naturally exploit such features. For example, we can add the production

```
Long↑a → - Long↑a Long↑c Four (↓c) IsTemp (↓a)
                         AUTODEC (↓'subl2'↓c↓a)
```

This production will be matched only when we find a subtraction of the constant 4 from a long format datum in a temporary. AUTODEC is a version of EMIT that delays generation of a subtraction instruction in hopes of realizing it as an auto-decrement in a future instruction. If this optimization cannot be done (or the updated value of the expression is needed), the subtraction is generated. Similarly, AUTOINC is another variant of EMIT that attempts to use auto-increment in an earlier instruction (Section 5.3).

In summary, the IR is translated to target code by parsing it through attribute-grammar productions. Multiple matches are handled by disambiguating predicates. A simple code generator can be implemented for a new machine using these basic productions. As time permits, the code generator can be tuned by adding new "optimization productions". The next chapter describes optimization productions and other machine-dependent optimizations.

Chapter 5: Machine-Dependent Optimization

Optimizing compilers attempt to produce a more efficient representation of user programs, aiming both for compact object code size (an operational constraint on computers with limited address space) and execution speed. A large number of optimizations are wholly architecture-dependent:
(1) using special machine instructions (e.g. increment, subtract-one-and-branch, add-one-and-branch-less-than-or-equal) and available addressing modes (e.g. indexing) to avoid explicit addition; also, subsuming addition or subtraction (e.g. using auto-increment/decrement),
(2) avoiding redundant register loads and stores (or redundant pushes and pops),
(3) optimizing branches (e.g. branch chaining, cross-jumping [Wulf 75], span-dependent instructions [Robertson 77, Szymanski 78, Szymanski 8Ø]). Code generation for control constructs has not been given much attention in recent approaches to formalization and automatic derivation of code generators. For example, Fraser, Glanville, Cattell and Ripken have not handled redundant condition code setting and the problems of generating short or long branches (as found in the PDP-11/7Ø, VAX-11/78Ø and many other mini and micro-computers). Glanville does not provide a means for describing long and short branches. Fraser and Cattell

are able to describe the restrictions imposed on short forms in ISP but they do not exploit this capability. In fact, all three imply the use of short forms in their examples. Ripken uses attribute predicates to describe different forms of branch instructions, including forms other than merely long and short. (The VAX-11/780 has three types of branch instructions: branch-byte (2 bytes), branch-word (3 bytes) and jump (5 bytes)). He proposes rearranging basic blocks of code in order to generate appropriate branch instructions in a later pass. Finding such an optimal rearrangement (to minimize program length) has been proved to be NP-complete by Robertson and Szymanski.

(4) peephole optimizations [McKeeman 65]. A separate peephole-optimization pass over assembler code is used by the Unix C compiler [Ritchie 78]. Bliss' FINAL [Wulf 75] has demonstrated that a considerable reduction (15-40%) in code size can be achieved by such a pass. FINAL directs almost all its efforts in improving code for control constructs. Recently, Fraser [Fraser 79, Fraser 80] has implemented a machine-independent peephole optimizer that reads machine descriptions and attempts to optimize adjacent pairs of assembler instructions. For a window of more than two instructions, the speed of the optimizer is degraded. Optimization of instructions not physically adja-

cent requires more 'context' information. Attributes are a good means of maintaining contextual information.

Our implementation attempts to express optimization in a non-procedural form, replacing the hand-coding of machine-dependent optimizations by the use of attribute grammars. Our intention is not to expand on the vast store of optimization techniques, but to cleanly organize 'tricky' machine-dependent optimizations (especially those optimizations that are both popular and effective). Some of these optimizations, such as removing redundant loads/stores and using arithmetic shifts instead of multiplications, are commonly used in compilers with the help of specially hand-coded routines. Others, such as the use of 'sob' (subtract-one-and-branch) on the PDP-11/70 and auto-increment, are not common. Our implementation formalizes machine-dependent optimization within the attributed parsing framework under the following categories:

(1) addition of attribute grammar productions to incorporate special instructions,

(2) delaying generation of code till the end of a basic block,

(3) code subsumption within addressing modes,

(4) deletion of redundant code and

(5) code alteration (back-patching) using information gathered after instruction selection.

5.1 Handling Special Instructions

In order to use special instructions, productions are added to describe special combinations of

(1) operands (e.g. multiplication or division of an integer by a power of two, incrementing or decrementing integers by one, assignments to or from an operand at the stack top) and

(2) operations (e.g. subtract-one-and-branch on the PDP-11/70, subtract-one-and-branch-greater-than and add-one-and-branch-less-than-or-equal on the VAX-11/780).

These optimization productions are added incrementally to improve the target code. They are specified before general productions so that their predicates are checked first. Such a scheme enhances modularity and allows incremental development of a code generator.

Consider the addition of another disambiguating predicate DontTrySob before IsTemp in the Decrement production (Section 4.4) to obtain

```
Long↑a → - Long↑a Long↑b IsOne (↓b) DontTrySob (↓a)
                         IsTemp (↓a) EMIT (↓'decl' ↓a)
```

In this case, both DontTrySob and IsTemp must evaluate to true in order that the production be applicable. The context in which an evaluation takes place is determined by interrogating the left context of the evaluation (i.e.

looking at symbols below the RHS on the stack). In the assignment context ":= A - B 1", the target address A is known after step (1) in Section 4.4. If "a" here is not the same as A (a global attribute keeps track of A) or if the following operator (which is determined by examining the look-ahead already provided by the parser) is not a 'greater-than' or 'greater-than-or-equal' comparison with zero, then DontTrySob evaluates to true. This evaluation does not include any 'less-than' comparisons because the VAX-11/780 does not provide a corresponding special instruction for 'subtract-one-and-branch'. If DontTrySob evaluates to false, recognition of this production is blocked in hopes of matching one of the following longer productions so that a special instruction may be selected. If this hope fails (determined by the predicate SobOk), then an alternate longer production is matched and a sequence of equivalent instructions is emitted. In the following example, 'Ø<' is a unary operator that tests if its operand is greater than zero. Similarly, 'Ø=<' tests if its operand is greater than or equal to zero. 'Ørel' stands for other tests with zero as the second operand:

```
Instruction → := Long↑d - Long↑a Long↑b Ø< Long↑c Label↑n
                         IsOne (↓b) SobOk (↓d↓a↓c↓n)
                         EMIT (↓'sobgtr' ↓d ↓n)

            → := Long↑d - Long↑a Long↑b Ø=< Long↑c Label↑n
                         IsOne (↓b) SobOk (↓d↓a↓c↓n)
                         EMIT (↓'sobgeq' ↓d ↓n)

            → := Long↑d - Long↑a Long↑b Ørel↑branch Long↑c
                         Label↑n IsOne (↓b)
                                 EMIT (↓'decl' ↓d)
                                 EMIT (↓'tstl' ↓c)
                                 EMIT (↓branch ↓n)
```

The disambiguating predicate SobOk evaluates to true if c, d and a are the same variable, and the distance of n from the current position is not greater than the short-branch distance. Forward references are taken as farther than a short-branch distance. If SobOk evaluates to false, we generate the sequence of instructions that would have been generated if DontTrySob evaluated to true. This sequence consists of first decrementing d, then setting condition codes by testing c and finally branching on the condition of the appropriate condition-code bit(s).

5.2 Delaying Code Generation

Within basic blocks [Aho 77], variables should be kept in faster locations as much as possible. Our code generator attempts to replace references to a variable's memory address by equivalent but cheaper addressing modes. The principles on which this strategy is based are:

(1) Whenever an assignment involves moving an operand from a cheaper addressing mode to a costlier one, the generation of the move instruction is delayed. Thus, in step (6) of 4.4, the movement of r_1 to A is delayed.

(2) Operand relocations (Section 3.3) may involve movement from a costlier addressing mode to a cheaper mode (e.g. moving B to r_1 in step (3) of 4.4). In such cases, the move instruction is 'hoisted' to the position after the last use of the cheaper addressing mode (the last use of r_1 in the preceding example) within the current basic block and after all intervening assignments to B. Then all subsequent references to B are replaced by r_1.

Such hoisting of register loads and subsequent alterations of addressing modes are achieved through the use of data structures that provide the necessary flexibility to alter operand addressing: instructions that are buffered use descriptors for the addressing modes of their operands. These addressing alterations require buffering code for the duration of a basic block. Instead of a straight-forward 'match-generate-match-generate' code-generation scheme, many matches are performed followed by a single 'generate' at the end of a basic block. Such optimizations assume that if an instruction can take an expensive addressing mode, it can also take a cheaper one. On machines with non-orthogonal instruction sets where this assumption might not prevail, this optimization should not be done. Therefore, our implementation permits optional use of this optimization.

5.3 Subsuming Code

Attributes are used to buffer previously generated instructions in order to subsume additions and subtractions by addressing modes such as

(1) indexing:

If the subtraction in step (5) of 4.4 was performed within the context of address calculation (i.e the IR representation is := A @ - B 1), a longer production is matched:

$$\text{Addr}\uparrow a \rightarrow @ - \text{Long}\uparrow b\ \text{Long}\uparrow c\ \underline{\text{IsCons}}\ (\downarrow c)\ \underline{\text{IsReg}}\ (\downarrow b)$$
$$\overline{\text{ADDR}\ (\downarrow -\ \downarrow b\ \downarrow c\ \uparrow a)}$$

If the attribute variable "c" is an integer constant and "b" is a register, then the subtraction is implicitly performed by using the index addressing mode supported by the architecture. ADDR composes an address attribute "a" with negative displacement "c" and base register "b".

(2) auto-increment/decrement:

The subtraction from "a" (say r_2) in Section 4.4 was delayed in hopes of realizing it as a future auto-decrement. If the next use of r_2 is an indirect reference through r_2 and the operation uses long data, then the auto-decrement addressing mode for r_2 is issued for the current instruction. Similarly, auto-increment modes are used to subsume additions to a register (using buffered information). If a register is incremented by a constant that is usable in an auto-increment, the previous use of the register addressing-mode is altered to the auto-increment mode using the mechanism outlined in the previous section.

5.4 Deleting Redundant Code

Buffering code (Section 5.2) and maintaining lists of equivalent addresses also help avoid redundant loads and stores. Other examples of deleted redundant code are branches to the immediately following instruction and unnecessary tests that precede branches. For example, if a comparison with zero follows a decrement instruction, then the following production is matched:

```
Cc↑br   → Ørelop↑br Long↑a
                              IF NOT Ccset THEN
                              EMIT (↓'tstl' ↓a)
```

The test instruction is not emitted because the condition codes are set correctly by the preceding decrement. More generally, sometimes an instruction is used only for setting condition codes. If its execution would set condition codes exactly as the preceding instruction did, then the instruction is suppressed.

5.5 Back Patching

Many architectures provide more than a single op-code for an unconditional branch. In our implementation, these op-codes are specified with their branch distances, e.g. for the VAX-11/780:

```
#define BRB                 "brb"    /* shortest branch      */
#define BRW                 "brw"    /* branch word          */
#define JMP                 "jmp"    /* longest branch(jump) */
#define SHORTDIST           254      /* byte-branch distance */
#define WORDDIST            32766    /* word-branch distance */
```

The exact forms of branch instructions can usually be determined only after their targets are defined. For backward branches, the exact form can be determined when code is generated. In the case of forward branches, the longest available form is used and, once the target is determined, the correct form is substituted. This strategy allows us to handle multiple forms of branch instructions within a single pass. Many machines (e.g. the PDP-11/70, VAX-11/780), only have conditional branches of the short form. Long-form conditionals are therefore simulated using a three-instruction sequence. For example, on a forward reference on the VAX-11/780, conditional branches (say 'cbranch Label') are converted into the sequence:

```
        opposite-of-cbranch(condition code)  internal-label
        jmp    Label
internal-label:
```

The opposite branch-opcode of a conditional branch is supplied by the grammar (e.g. the opposite branch of branch-if-greater, 'bgtr', on the VAX-11/780 is branch-if-less-than-or-equal-to, 'bleq'). The code generator generates necessary internal labels. When Label is subsequently defined, a cheaper form of branch instruction is used. The exact branch distance is calculated; if the distance is less than the short-branch distance, the entire three-instruction sequence is replaced by 'cbranch Label'. If the distance is representable in a branch-word instruction, the "jmp" is altered to "brw".

5.6 Time versus Space Optimizations

Attributes and disambiguating predicates are very useful in choosing between optimization for space and optimization for time when they conflict. For example, the VAX-11/780 has a three-address arithmetic shift (ashl) and both two and three-address multiply instructions (mull2, mull3) that we can exploit:

```
Long↑r → * Long↑a Long↑r IsTemp (↓r)
          TimeOpt (↓'ashl'↓'mull2') PowerTwo (↓a) LOG2 (↓a↑p)
                              EMIT (↓'ashl' ↓p ↓r ↓r)

       → * Long↑r Long↑a IsTemp (↓r)
          TimeOpt (↓'ashl'↓'mull2') PowerTwo (↓a) LOG2 (↓a↑p)
                              EMIT (↓'ashl' ↓p ↓r ↓r)

       → * Long↑r Long↑a TwoOp (↓*↓a↓r)
                              EMIT (↓'mull2' ↓a ↓r)

       → * Long↑a Long↑r TwoOp (↓*↓a↓r)
                              EMIT (↓'mull2' ↓a ↓r)

       → * Long↑a Long↑b PowerTwo (↓a) LOG2 (↓a ↑p)
                              GETTEMP (↓'long' ↑r)
                              EMIT (↓'ashl' ↓p ↓b ↓r)

       → * Long↑a Long↑b PowerTwo (↓b) LOG2 (↓b ↑p)
                              GETTEMP (↓'long' ↑r)
                              EMIT (↓'ashl' ↓p ↓a ↓r)

       → * Long↑a Long↑b
                              GETTEMP (↓'long' ↑r)
                              EMIT (↓'mull3' ↓a ↓b ↓r)
```

The disambiguating predicate PowerTwo evaluates to true if its attribute is a power of two. The applicability of the arithmetic shift production depends further on whether optimization for time is to be preferred over optimization for space. A two-address multiply on the VAX-11/780 occupies less space than an arithmetic shift (which needs three addresses). But an arithmetic shift is considerably faster than a multiply instruction. The compiler writer may define predicates such as TimeOpt in terms of options that are set by the user during compilation.

Chapter 6: Implementation and Results

Implementations of our code generator exist on both the PDP-11/70 and the VAX-11/780. The generator occupies 86K bytes on the PDP-11/70 (37K text + 45K data + 4.3K uninitialized data) and 115K bytes on the VAX-11/780 (48K text + 63K data + 3.9K uninitialized data). The 11/70 implementation generates about 30 lines of assembler code per second (real time), while the VAX version generates about 50 lines per second. In contrast, the C compiler produces about 40 lines per second on the PDP-11/70 and about 60 lines per second on the VAX-11/780. The goals of the implementation are:

(1) portability (minimum change required to retarget the code generator to new machines),

(2) use of state-of-the-art techniques (attribute grammars and attributed parsing),

(3) efficiency of code generation (a one-pass linear parsing technique),

(4) flexibility (code optimizations incrementally incorporated, all optimizations optional),

(5) modularity (instruction-set specification by addition of new productions).

Producing a small code generator was not among the primary goals. Nevertheless, the size turned out to be reasonable.

The code generator can be compiled and used on a minicomputer such as the PDP-11/70.

YACC [Johnson 75], running on the PDP-11/70 and the VAX-11/780, was used to generate tables for these implementations. Its driver was modified to accommodate disambiguating predicates and create parsers for attribute grammars. It required about four minutes to process each of the (highly optimizing) code-generation grammars. More detailed statistics show the complexity of the grammar required:

```
PDP-11/70 grammar:
        55 terminals
        150 non-terminals
        346 grammar rules
        739 parser states
        time taken on the PDP-11/70     = 3.48 minutes
                        (real time      = 3.48 minutes,
                         user time      = 2.05 minutes,
                         system time    = 8.8 seconds)
VAX-11/780 grammar:
        63 terminals
        179 non-terminals
        578 grammar rules
        1273 parser states
        time taken on the VAX-11/780    = 4.09 minutes
                        (real time      = 4.09 minutes,
                         user time      = 4.01 minutes,
                         system time    = 3.8 seconds)
```

The code produced by our implementations (Cg) on the VAX-11/780 and the PDP-11/70 was compared with that produced by C compilers for the binary-search and string-comparison programs given in Section 2.4. We will ignore assembler code for allocating space for variables, because it is not important for purposes of comparison. Cg produced the following VAX-11/78Ø code for the binary search IR (procedure parameters are pushed in reverse order):

```
                                        # begin level 1
b_search:                               # begin level 2
        subl2   $12, sp
        movl    $1, -4(fp)              # := low 1
        movl    $1Ø, -8(fp)             # := high 1Ø
        moval   item, r1                # := adritem #item
L2Ø:    addl3   -4(fp), -8(fp), r2
        divl2   $2, r2                  # := middle r2
        ashl    $2, r2, r3              # * middle SIZE
        addl2   r1, r3                  # + adritem r3
        cmpl    (r3), 4(ap)
        blss    L21
        addl3   r2, $1, -4(fp)          # + middle 1
L21:    cmpl    (r3), 4(ap)
        bgtr    L19
        subl3   $1, r2, -8(fp)          # - middle 1
L19:    cmpl    -4(fp), -8(fp)          # cmp low high
        bleq    L2Ø
        addl3   -4(fp), $1, r4          # + low 1
        cmpl    r4, -8(fp)              # cmp r4 high
        bleq    L23
        movl    r2, rØ                  # := bsrch middle
        brb     proced_end
L23:    clrl    rØ                      # := b_search Ø
proced_end:
        ret                             # end level 2
_main:
        movl    $1, index               # := index 1
        addl3   $item, $4, rØ           # + #item SIZE
L3Ø:    movl    index, (rØ)+            # repeat scope
        aobleq  $1Ø, index, L3Ø         # inc & test
        pushl   $5                      # arg for bsrch
        calls   $1, b_search            # proc call
        movl    rØ, result              # := result proc
        beql    L31
        pushl   result                  # arg for print
        calls   $1, _printd             # proc call
L31:    ret                             # end level 1
```

Three-address instructions have been used optimally in lines 7, 13, 16 and 19. Since, by default, optimization for space is preferred to that for time, a two-address 'divl2' is used instead of an arithmetic shift in line 8. The 'compares' (lines 11 and 14) use register-indirect addressing mode for 'item[middle]'. The value in r_3-indirect is used in line 14. The value of 'middle' is saved in r_2 and is used in lines 16 and 22. The increment of 'index' in 'item[index]' within the repeat-loop is subsumed as an auto-increment of register $r_Ø$ (line 3Ø). Instead of an increment-compare-branch instruction sequence, the special instruction 'aobleq' is used in line 31. All branch instructions are of the shortest form. Possible improvements to this code are:

(1) subsuming array-index calculations within the index addressing mode of the VAX,

(2) replacing branches to a return instruction by return instructions and

(3) retaining temporaries in registers across basic blocks.

Code for array-index calculation is produced because it was explicitly specified in the IR. The VAX's index addressing mode avoids explicit index calculations. A special subscript operator could be added to the IR to capture this mode. In this example, using the index addressing mode would make the compare instructions occupy more space.

For comparison, the C compiler produced the following code for the same binary search program:

```
_b_searc:
          jbr     L16
L17:      movl    $1,-4(fp)
          movl    $10,-8(fp)
L20:      addl3   -8(fp),-4(fp),r0
          divl2   $2,r0
          movl    r0,r11
          cmpl    _item[r11],4(ap)
          jlss    L21
          addl3   $1,r11,r0
          movl    r0,-4(fp)
L21:      cmpl    _item[r11],4(ap)
          jgtr    L22
          subl3   $1,r11,r0
          movl    r0,-8(fp)
L22:L19:cmpl    -4(fp),-8(fp)
          jleq    L20
L18:      addl3   $1,-4(fp),r0
          cmpl    r0,-8(fp)
          jleq    L23
          movl    r11,r0
          ret
L23:      clrl    r0
          ret
          ret
L16:      subl2   $8,sp
          jbr     L17
_main:
          jbr     L26
L27:      movl    $1,_index
L30:      movl    _index,r0
          movl    _index,_item[r0]
          incl    _index
L29:      cmpl    _index,$10
          jleq    L30
L28:      pushl   $5
          calls   $1,_b_searc
          movl    r0,_result
          tstl    _result
          jeql    L31
          pushl   _result
          calls   $1,_printd
L31:      ret
L26:      jbr     L27
```

After the optimization pass (-O flag), the code is:

```
 1  _b_searc:
 2              subl2    $8,sp
 3              movl     $1,-4(fp)
 4              movl     $1Ø,-8(fp)
 5  L2Ø:        addl3    -8(fp),-4(fp),rØ
 6              divl3    $2,rØ,r11
 7              cmpl     _item[r11],4(ap)
 8              jlss     L21
 9              addl3    $1,r11,-4(fp)
1Ø  L21:        cmpl     _item[r11],4(ap)
11              jgtr     L19
12              subl3    $1,r11,-8(fp)
13  L19:        cmpl     -4(fp),-8(fp)
14              jleq     L2Ø
15              addl3    $1,-4(fp),rØ
16              cmpl     rØ,-8(fp)
17              jleq     L23
18              movl     r11,rØ
19              ret
2Ø  L23:        clrl     rØ
21              ret
22  _main:
23              movl     $1,_index
24  L3Ø:        movl     _index,rØ
25              movl     rØ,_item[rØ]
26              aobleq   $1Ø,_index,L3Ø
27              pushl    $5
28              calls    $1,_b_searc
29              movl     rØ,_result
3Ø              tstl     rØ
31              jeql     L31
32              pushl    rØ
33              calls    $1,_printd
34  L31:        ret
```

The unoptimized version of code generated by the C compiler occupies 37% more space than that produced by Cg. A two-address 'divl2' followed by a 'movl' is used instead of a three-address divide (lines 6 and 7). A three-address 'addl3' into $r_Ø$ followed by a 'movl' of $r_Ø$ into '-4(fp)' is used instead of a more optimal 'addl3' into '-4(fp)' (lines

1Ø and 11). Similarly in lines 14 and 15, the 'subl3' instruction is followed by a 'movl'. The index addressing mode on the VAX has been used to avoid generating code for subscript calculations (lines 8, 12 and 32). However, the 'aobleq' instruction is not used to optimize an increment-compare-branch sequence. Furthermore, all branch instructions are of unresolved length. The C compiler relies on the assembler to resolve long/short forms of branch instructions.

The optimized code uses three-address instructions optimally (lines 5, 9, 12 and 15). Furthermore, the special case 'aobleq' is recognized (line 26). After the optimization pass, the C compiler's code occupies 14% more space than that produced by Cg. Cg gets this advantage by remembering register contents, using the auto-increment addressing mode, and using short forms of branch instructions. The time taken by these programs is not large enough to be significantly compared (Ø.Ø seconds). The difference in speeds cannot be resolved by using the built-in clock. Ideally, a comparison could be made using instruction execution times published by the manufacturer but these figures are not released by Digital Equipment Corporation. Furthermore, issues such as cache usage may obscure such a comparison. Cg produced the following PDP-11/7Ø code for the binary search program:

```
                                          / octal constants
                                          / begin level 1
b_search:                                 / begin level 2
        jsr     r5, csv
        sub     $4, sp
        mov     $1, -10(r5)               / := low 1
        mov     $12, -12(r5)              / := high 10
        mov     $item, r1                 / := adritem #item
L20:    mov     -10(r5), r2
        add     -12(r5), r2
        asr     r2                        / := middle r2
        mov     r2, r3
        asl     r3                        / * middle SIZE
        add     r1, r3                    / + adritem r3
        cmp     (r3), 4(r5)
        blt     L21
        mov     r2, -10(r5)
        inc     -10(r5)                   / + middle 1
L21:    cmp     (r3), 4(r5)
        bgt     L19
        mov     r2, -12(r5)
        dec     -12(r5)                   / - middle 1
L19:    cmp     -10(r5), -12(r5) / cmp low high
        ble     L20
        mov     -10(r5), r4
        inc     r4                        / + low 1
        cmp     r4, -12(r5)
        ble     L23
        mov     r2, r0                    / := bsrch middle
        br      proced_end
L23:    clr     r0                        / := bsrch 0
proced_end:
        jmp     cret                      / end level 2
_main:
        mov     $1, index                 / := index 1
        mov     $item, r0
        add     $2, r0                    / + #item SIZE
L30:    mov     index, (r0)+              / repeat scope
        inc     index                     / increment
        cmp     index, $12                / test
        ble     L30
        mov     $5, (sp)                  / arg for bsrch
        jsr     pc, *$b_search
        mov     r0, result                / := result proc
        beq     L31
        mov     result, (sp)              / arg for print
        jsr     pc, *$_printd
L31:    jmp     cret                      /end level 1
```

The C compiler on the PDP-11/7Ø produced:

```
_b_searc:jsr     r5,csv
         jbr     L1
L2:      mov     $1,-1Ø(r5)
         mov     $12,-12(r5)
L6:      mov     -1Ø(r5),r1
         add     -12(r5),r1
         sxt     rØ
         div     $2,rØ                / could use shift
         mov     rØ,r4                / redundant move
         mov     r4,rØ                / redundant move
         asl     rØ
         cmp     4(r5),_item(rØ)
         jgt     L7
         mov     r4,rØ
         inc     rØ
         mov     rØ,-1Ø(r5)
L7:      mov     r4,rØ
         asl     rØ
         cmp     4(r5),_item(rØ)
         jlt     L8
         mov     r4,rØ
         dec     rØ
         mov     rØ,-12(r5)
L8:L4:   cmp     -12(r5),-1Ø(r5)
         jge     L6
L5:      mov     -1Ø(r5),rØ
         inc     rØ
         cmp     -12(r5),rØ
         jge     L9
         mov     r4,rØ
         jbr     L3
L9:      clr     rØ
         jbr     L3
L3:      jmp     cret
L1:      sub     $4,sp
         jbr     L2
_main:   jsr     r5,csv
         jbr     L1Ø
L11:     mov     $1,_index
L15:     mov     _index,rØ
         asl     rØ
         mov     _index,_item(rØ)
         inc     _index
L13:     cmp     $12,_index
         jge     L15
L14:     mov     $5,(sp)
         jsr     pc,*$_b_searc
         mov     rØ,_result
```

```
49          tst     _result        / redundant test
50          jeq     L16
51          mov     _result,(sp)
52          jsr     pc,*$_printd
53  L16:L12:jmp     cret
54  L10:    jbr     L11
```

The C compiler's code for the PDP-11/70 occupies 30% more space than that produced by Cg. Cg uses an arithmetic shift instruction in line 11 whereas the C compiler produced a divide instruction (line 8). Furthermore, lines 9 and 10 are redundant move instructions and line 49 is a redundant test produced by the C compiler. Cg did not produce these redundant instructions and was also able to employ an auto-increment in line 38. The code produced by the C compiler after its optimization pass is given on the next page. Once again, the arithmetic shift is not used instead of division by two, and the redundant move instructions are still present. However, the redundant test is removed. The C compiler's code now occupies 15% more space than that produced by Cg.

C compiler's code after optimization pass:

```
_b_searc:jsr     r5,csv
         sub     $4,sp
         mov     $1,-10(r5)
         mov     $12,-12(r5)
L6:      mov     -10(r5),r1
         add     -12(r5),r1
         sxt     r0
         div     $2,r0              / could use shift
         mov     r0,r4              / redundant move
         mov     r4,r0              / redundant move
         asl     r0
         cmp     4(r5),_item(r0)
         jgt     L7
         mov     r4,r0
         inc     r0
         mov     r0,-10(r5)
L7:      mov     r4,r0
         asl     r0
         cmp     4(r5),_item(r0)
         jlt     L4
         mov     r4,r0
         dec     r0
         mov     r0,-12(r5)
L4:      cmp     -12(r5),-10(r5)
         jge     L6
         mov     -10(r5),r0
         inc     r0
         cmp     -12(r5),r0
         jge     L9
         mov     r4,r0
L3:      jmp     cret
L9:      clr     r0
         jbr     L3
_main:   jsr     r5,csv
         mov     $1,_index
L15:     mov     _index,r0
         asl     r0
         mov     _index,_item(r0)
         inc     _index
         cmp     $12,_index
         jge     L15
         mov     $5,(sp)
         jsr     pc,*$_b_searc
         mov     r0,_result
         jeq     L12
         mov     r0,(sp)
         jsr     pc,*$_printd
L12:     jmp     cret
```

We now compare the code produced for the string-comparison program of Section 2.4. This program was intentionally chosen as a case in which the C compiler could produce highly optimized code. Cg produced the following VAX-11/78Ø code:

```
_main:                                  # begin level 1
             movl    8(ap), r1          # argv in a reg
             cmpl    4(ap), $2          # cmp argc 2
             bleq    L15
             movl    4(r1), rØ          # arg ← argv+4
             cmpb    (rØ), $45          # ascii '-'
             bneq    L2
             cmpb    1(rØ), $112        # ascii 'p'
             bneq    L2
             pushl   $8       # last parameter first
             pushl   12(r1)   # second parameter
             pushl   8(r1)    # first parameter
             calls   $3, strncmp        # proc call
             tstl    rØ
             bneq    L3
             pushl   $1       # parameter for exit
             calls   $1, exit
L3: L2:  L15:
             pushl   $Ø       # parameter for exit
             calls   $1, exit
             ret                        # end level 1
strncmp:                                # begin level 1
             movl    12(ap), r5         # register pref
             movl    8(ap), r4
             movl    4(ap), r3
             clrl    r1                 # := i Ø
             brb     L25
L2ØØ1:       clrb    r2       # explicit Boolean result
             cmpb    (r3)+, (r4)+
             bneq    L9ØØØ
             incb    r2
L9ØØØ:       tstb    r2
             bneq    L23
             movl    $1, rØ             # := strncmp 1
             brb     L13
L23:         incl    r1                 # := i + i 1
L25:         cmpl    r1, r5             # cmp i len
             blss    L2ØØ1
             clrl    rØ                 # := strncmp Ø
L13:         ret                        # end level 1
```

Cg used indexing (i.e. 1(rØ)) instead of addition by 1 (line 8) and auto-increments, i.e., cmpb (r3)+, (r4)+ (line 29). However, because of the IR for 'if(*str1++ != *str2++)', r2 is used to store the result of the comparison. After optimization, the C compiler produced the following VAX-11/78Ø code:

```
 1  _main:
 2           movl     8(ap),r11
 3           cmpl     4(ap),$2
 4           jleq     L15
 5           movl     4(r11),r1Ø
 6           cmpb     (r1Ø),$45
 7           jneq     L15
 8           cmpb     1(r1Ø),$112
 9           jneq     L15
1Ø           pushl    $8
11           pushl    12(r11)
12           pushl    8(r11)
13           calls    $3,_strncmp
14           tstl     rØ
15           jneq     L15
16           pushl    $1
17           calls    $1,_exit
18  L15:     pushl    $Ø
19           calls    $1,_exit
2Ø           ret
21  _strncmp:
22           movl     4(ap),r11
23           movl     8(ap),r1Ø
24           movl     12(ap),r9
25           clrl     r8
26           jbr      L25
27  L2ØØØ1:  cmpb     (r11)+,(r1Ø)+
28           jeql     L23
29           movl     $1,rØ
3Ø           ret
31  L23:     incl     r8
32  L25:     cmpl     r8,r9
33           jlss     L2ØØØ1
34           clrl     rØ
35           ret
```

The C compiler's code after optimization occupies 10% more space than that produced by Cg (mainly due to branch instructions). Cg produced the following PDP-11/70 code:

```
_main:                                 / begin level 1
            jsr     r5, csv
            mov     6(r5), r1          / argv in a reg
            cmp     4(r5), $2          / cmp argc 2
            ble     L15
            mov     2(r1), r0          / arg ← arg+4
            cmpb    (r0), $55          / ascii '-'
            bne     L2
            cmpb    1(r0), $160        / ascii 'p'
            bne     L2
            mov     $10, (sp)          / last parameter
            mov     6(r1), -(sp)       / second parameter
            mov     4(r1), -(sp)       / first parameter
            jsr     pc, *$strncmp
            cmp     (sp)+, (sp)+       / reset stack top
            tst     r0
            bne     L3
            mov     $1, (sp)           / parameter, exit
            jsr     pc, *$exit
L3: L2:  L15:
            clr     (sp)               / parameter, exit
            jsr     pc, *$exit
            jmp     cret               / end level 1
strncmp:                               / begin level 1
            jsr     r5, csv
            mov     10(r5), (sp)       / register pref
            mov     6(r5), r4
            mov     4(r5), r3
            clr     r1                 / := i 0
            br      L25
L2001:      clrb    r2       / explicit Boolean result
            cmpb    (r3)+, (r4)+
            bne      L9004
            incb    r2
 L9004:     tstb    r2
            bne     L23
            mov     $1, r0             / := strncmp 1
            br      L13
L23:        inc     r1                 / := i + i 1
L25:        cmp     r1, (sp)           / cmp i len
            blt     L2001
            clr     r0                 / := strncmp 0
L13:        jmp     cret               / end level 1
```

The C compiler produced the following PDP-11/70 code (after optimization pass):

```
_main:
          jsr      r5,csv
          mov      6(r5),r4
          cmp      $2,4(r5)
          jge      L4
          mov      2(r4),r3
          cmpb     $55,(r3)
          jne      L4
          cmpb     $160,1(r3)
          jne      L4
          mov      $10,(sp)
          mov      6(r4),-(sp)
          mov      4(r4),-(sp)
          jsr      pc,*$_strncmp
          cmp      (sp)+,(sp)+
          tst      r0
          jne      L4
          mov      $1,(sp)
          jsr      pc,*$_exit
L4:       clr      (sp)
          jsr      pc,*$_exit
          jmp      cret
_strncmp:
          jsr      r5,csv
          mov      4(r5),r4
          mov      6(r5),r3
          mov      10(r5),r2
          tst      -(sp)
          clr      -10(r5)
          jbr      L10
L20001:   cmpb     (r3)+,(r4)+
          jeq      L12
          mov      $1,r0
L9:       jmp      cret
L12:      inc      -10(r5)
L10:      cmp      r2,-10(r5)
          jgt      L20001
          clr      r0
          jbr      L9
```

Even though the C compiler uses two passes, its output occupies 11% more space than that produced by Cg. In most other respects, the two programs are almost identical.

Chapter 7: Conclusion

Language development tools on the Unix system [Johnson 80] have been used to process attribute grammars for purposes of automating compiler code generation. The availability of attribute-grammar parsers could have considerably eased and hastened this implementation. Nevertheless, YACC and LEX were very useful. Without these tools, it would have been harder and more time-consuming to implement the code generator.

Earlier research done by Glanville has been extended by adding attributes to instruction-set descriptions. The resulting code generator has demonstrated a definite improvement in code quality over Glanville's code generator. Machine-dependent optimizations such as using specialized instructions, complex addressing modes (e.g. auto-increment/decrement), span-dependent branch optimizations and peephole optimizations over a very wide window have been incorporated within the attributed parsing framework of code generation. Such optimizations have defied automatic code-generators of the past. Furthermore, all these optimizations are essentially obtained in a single-pass code generation scheme. In most cases, the results reveal better code than that produced by the C compiler using its additional pass of peephole optimization. The time taken by Cg to produce code is roughly the same as that taken by the entire C compiler. This observation suggests that when Cg is used with a compiler front end, the total time taken will be more than that of the C compiler. However, experimental results suggest that the front end need not take more than a few seconds to produce the IR required by Cg on reasonably large programs. Moreover, it is far easier to interface front ends with Cg than with the C compiler's code generator. It required less than ten hours to produce attributed prefix IR from a Modula front end, whereas it took months to interface a Pascal front end with the Portable C compiler's code generator.

The code-optimization results are very encouraging. Cg produces code comparable to hand-written assembler code for user programs. It produces code far superior to the unoptimizing C compiler on both the PDP-11/70 and the VAX-11/780. In most cases, Cg produces code that uses 35-50% less space than the code produced by the C compilers prior to "-O" optimization. This space reduction in object code is mainly due to optimization of redundant loads and stores, using auto-increment/decrement addressing modes, using specialized instructions and short forms of branch instructions wherever possible. In the examples used for comparison, C was given the advantage of explicit register preference declarations. If such explicit declarations are omitted, the C compiler makes no attempt to retain variables or results in registers. In contrast, Cg attempts to retain temporary results and variables in registers within basic blocks. This optimization is another reason for Cg's object code efficiency when compared with the C compiler. Even with the peephole optimization performed by the C compiler, the code produced by Cg is usually 5-10% smaller. The string-comparison example was chosen to illustrate a special case where the C compiler produces auto-increment addressing modes. The C compiler specifically looks for the auto-increment operator ('++') in a user program and never uses auto-increment otherwise. Therefore, if the C compiler's code generator is used with a Pascal front end, for example, it can never utilize the auto-increment addressing mode. In contrast, Cg recognizes general constructs amenable to auto-increment and auto-decrement. Most of its optimization power is derived from keeping results in registers, remembering equivalent locations and last usages of registers. It never emits redundant loads and stores, which are common even in the C compiler's optimized code. The C compiler always produces long-form branch op-codes. Resolution of long/short-address branch op-codes is left to the assembler. If the assembler does not perform span-dependent optimization, the code produced may be significantly larger.

An amazingly wide variety of code-generation optimizations can be realized in a highly modular manner. Almost all optimizations can be realized by addition of new attribute grammar productions. Furthermore, when the code generator is retargeted to a new machine, most of the basic (non-specialized) productions can be retained. In particular, a simple (but un-optimized) code generator can be implemented for a machine easily and rapidly. As time permits and the need arises, improvements can be included by adding new rules to the machine description and automatically re-generating the code generator. The chief difference between an optimized and an unoptimized code generator is how carefully and thoroughly the production rules reflect the details and complexities of the target machine.

We have retained the speed of Glanville's code generator by using a one-pass, linear parsing technique. It is faster than those implemented by Fraser and Cattell and is expected to be much faster than that proposed by Ripken. Furthermore, all properties established by Glanville hold for our implementation:

(1) correctness of the code generation algorithm,

(2) detection of syntactic errors in the IR, and

(3) detection of incomplete instruction-set specification by blocking (instead of infinitely looping or generating incorrect code).

Even though the code generator requires substantial data structures for optimization, its size is very reasonable. It can run on computers with a limited address space such as the PDP-11/7Ø.

Unlike Glanville and Cattell, we have viewed storage binding as part of the issue of portable code generation. Our design has attempted to isolate almost all machine-dependent aspects of compiler code generation to a single software package. In our code generator, it is very easy to alter activation record formats on the run-time stack, whereas in conventional compilers, this change may affect several sections of code. The intermediate representation designed here is at a higher level than that proposed by Glanville. Furthermore, use of attributes in the IR helps convey information from machine-independent global optimizers to the code generator. This design, therefore, provides a better interface with the machine-independent parts of a compiler and significantly simplifies retargeting all aspects of code generation to new machines.

Apart from its use in portable compilers, the code generator may be used in research to provide an easy technique for experimenting with various optimizations. Such experiments usually only require modification of the attribute grammar specification for the target machine.

This research has successfully demonstrated:

(1) the design of an IR for portable code generation,

(2) the use of attributes to interface machine-independent aspects of a compiler with the machine-dependent parts,

(3) the use of attribute grammars to describe the details and complexities of the target machine for purposes of code generation and

(4) the incorporation of machine-dependent and peephole optimizations in a routine, cheap and reliable manner within an attributed parsing framework of code generation.

There is considerable scope for improving the current implementation and for extending this research. The size of the code generator can be reduced to some extent by compacting the transition tables produced by LEX (which occupy 3ØK bytes). The source code can be further optimized (by hand) to reduce the size of the code generator.

The IR specification provides considerable flexibility in the form of attributes. As an extension, dynamic arrays, dope-vector specification, records, Simula classes and Modula processes can be added to our design. Records and structure fields can be specified by the front end (as '. R f', where 'R' is the name of the structure (record), 'f' is its field name and '.' is a qualification operator).

Instead of freeing all registers at the end of a basic block, the code generator can be guided by global analysis of variables that are live or dead upon exit from a basic block. In the binary search program, the last 'pushl' instruction could use less space if 'rØ' were used instead of 'result'. Before exiting the basic block that ends with 'beql L31', the code generator could have retained the value of 'result' in 'rØ' (with some guidance from the front end).

All these extensions can be easily incorporated using the data structures and routines existing in our implementation. Our implementation turned out to be modularly divided into the storage binding phase (done by LEX) and the instruction selection phase (done by YACC). The extensions proposed above are restricted to the LEX part of this implementation.

We have experimented with the code generator as part of a real compiler in only a few test runs. More experiments are necessary to test the code generator in real compiler environments. Although we have experimented only with the VAX-11/78Ø and the PDP-11/7Ø, it should be fairly easy to retarget the code generator to the IBM-37Ø by adding attributes to Glanville's existing instruction-set specification for the IBM-37Ø. However, more experiments with attribute grammar specifications are needed for other machines, including special purpose computers such as the Burroughs B-5500, CDC-Star, Cray-1, data-flow architectures and capability-based machines (Intel-iAPX 432).

The reader familiar with optimization literature will notice dozens of machine-dependent optimizations that are not mentioned in this dissertation. Our intention has not been to incorporate all possible optimizations, but rather to show how certain difficult ones can be easily incorporated in our code generation scheme. Some optimizations are dependent on others and can be applied iteratively. Thus, they suggest more than a single pass over the generated code. Time-varying attributes [Skedzeleski 78] can easily specify iterative algorithms in a non-procedural manner and efficiently implement them. Future research in iterative optimizations can therefore proceed in two directions:
(a) hiding iterations within attribute action-symbols, or
(b) introducing an attribute-evaluator iterator.

Such an iterative evaluation must be performed by traversing parse trees, but the order of evaluation of attributes can be calculated at evaluator-generation (code-generator generation) time rather than at code-generation time. The drawback of such an iterative approach may very well be the amount of time taken to achieve the optimizations. More research is necessary to determine if the time taken justifies the extra optimizations gained.

Optimization of object code to increase execution speed has not received much attention so far in code-optimization research. The use of disambiguating predicates (like the predicate TimeOpt in Chapter 5) with a separate ordering of these productions for time optimization could be a starting point for this research.

In this dissertation, we have investigated the use of attribute grammars in automating software (i.e. a code generator). By augmenting our attribute grammar specifications with more details (such as bit-level specifications) of the target architecture, they may become useful tools in hardware design and synthesis as an improvement to ISPL [Barbacci 76].

Bibliography

[Aho 73] A.V. Aho and J.D. Ullman, "The Theory of Parsing, Translation and Compiling", Vols. 1 and 2, Prentice-Hall, Inc., 1973.

[Aho 75] A.V. Aho, S.C. Johnson and J.D. Ullman, "Deterministic Parsing of Ambiguous Grammars", CACM Vol.18 No. 8, 1975.

[Aho 76] A.V. Aho and S.C. Johnson, "Optimal Code Generation for Expression Trees", JACM Vol. 23 No. 3 pp. 488-5Ø1, 1976.

[Aho 77] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley publishing Co., 1977.

[Allen 72] F.E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations", in Design and Automation of Compilers, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Ammann 77] U. Ammann, "On Code Generation in a Pascal Compiler", Software-Practice and Experience,Vol. 7 No. 3 pp. 391-423, June/July 1977.

[Barbacci 76] M.R. Barbacci, "The ISPL Compiler and Simulator User's Manual", Tech. Report, Computer Science Dept., Carnegie-Mellon University, 1976.

[Barbacci 77] M.R. Barbacci, G.E. Barnes, R.G. Cattell, D.P. Siewiorek, "The ISPS Computer Description Language", Tech. Report, Dept. of Computer Science, Carnegie-Mellon University, 1977.

[Bell 71] C.G. Bell and A. Newell, "Computer Structures: Readings and Examples", McGraw Hill, 1971.

[Chu 74] Y. Chu, "Why Do We Need Computer Hardware Description Languages?", IEEE Computer, Vol. 7, No. 12, 1974.

[Coleman 73] S.S. Coleman, P.C. Poole and W.M. Waite, "The Mobile Programming System: Janus", National Tech. Infor. Center PB220322, U.S. Dept. of Commerce, Springfield, Va., 1973.

[Cattell 78] R.G.G. Cattell, "Formalization and Automatic Derivation of Code Generators", PhD thesis, Carnegie Mellon University 1978.

[Cattell 79] R.G.G. Cattell, J.M. Newcomer and B.W. Leverett, "Code Generation in a Machine-Independent Compiler", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.

[Cattell 80] R.G.G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions", ACM Trans. Programming Languages and Systems, Vol. 2 No. 2 pp. 173-190, April 1980.

[Dietmeyer 68] D.L. Dietmeyer and J.R. Duley, "A Digital System Design Language (DDL)", IEEE Transactions on Computers, C-17, 9, 1968.

[Dietmeyer 74] D.L. Dietmeyer, "Introducing DDL", IEEE Computer, Vol. 7, No. 12, 1974.

[Dietmeyer 78] D.L. Dietmeyer, "Logic Design of Digital Systems", Allyn and Bacon, 1978.

[Dijkstra 60] E.W. Dijkstra, "Algol 60 Translation", Supplement, Algol 60 Bulletin 10, 1960.

[Donegan 73] M.K. Donegan, "An Approach to the Automatic Generation of Code Generators", PhD thesis, Rice University, Houston, Texas, 1973.

[Donegan 79] M.K. Donegan et al., "A Code Generator Language", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.

[Elson 70] M. Elson and S.T. Rake, "Code Generation Technique for Large Language Compilers", I.B.M. Systems Journal Vol. 9 No. 3 pp. 166-188, 1970.

[Feustal 73] E.A. Feustal, "On the Advantages of Tagged Architecture", IEEETC, Vol. C-22 No. 7, July 1973.

[Fischer 8Ø] C.N. Fischer, D.R. Milton and S.B. Quiring, "Efficient LL(1) Error Correction and Recovery Using only Insertions", Acta Informatica Vol. 13, 198Ø.

[Frailey 7Ø] D. Frailey, "Expression Optimization Using Unary Complement Operators", ACM Sigplan Notices, 5, 197Ø.

[Fraser 77] C.W. Fraser, "Automatic Generation of Code Generators", PhD thesis, Computer Science Dept., Yale University, New Haven, Conn., 1977.

[Fraser 79] C.W. Fraser, "A Compact Machine Independent Peephole Optimizer", Principles Of Programming Languages, 1979.

[Fraser 8Ø] C.W. Fraser and J.W. Davidson, "The Design and Application of a Retargetable Peephole Optimizer", ACM Transactions on Programming Languages and Systems, Vol. 2 No. 2, 198Ø.

[Ganapathi 8Ø] M. Ganapathi and C.N. Fischer, "A Review of Automatic Code Generation Techniques", Tech. Report #4Ø6, University of Wisconsin - Madison, 198Ø.

[Glanville 77] R.S. Glanville, "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers", PhD thesis, University of California, Berkeley, Dec. 1977.

[Glanville 78] R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation", Conf. Record Fifth ACM Symp. Principles of Programming Languages, Jan. 1978.

[Graham 79] S.L. Graham, "Practical LR Error Recovery", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.

[Graham 8Ø] S.L. Graham, "Table-Driven Code Generation", IEEE Computer, Vol. 13 No. 8 pp. 25-34, Aug. 198Ø.

[Gries 71] D. Gries, "Compiler Construction for Digital Computers", John Wiley & Sons, 1971.

[Hill 74] F.J. Hill, "Introducing AHPL", IEEE Computer, Vol. 7, No. 12, 1974.

[Johnson 75] S.C. Johnson "YACC - Yet Another Compiler Compiler", C.S. Tech Report #32, Bell Telephone Laboratories, Murray Hill, New Jersey, 1975.

[Johnson 77] S.C. Johnson, "A Tour through the Portable C Compiler", Bell Telephone Laboratories, 1977.

[Johnson 78] S.C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symp. Principles of Programming Languages, pp. 97-104, Jan 1978.

[Johnson 80] S.C. Johnson, "Language Development Tools on the Unix System", IEEE Computer Vol. 13 No. 8 pp. 16-21, Aug. 1980.

[Johnsson 75] R.K. Johnsson, "An Approach to Global Register Allocation", PhD dissertation, Carnegie-Mellon University, 1975.

[Kennedy 71] K. Kennedy, "A Global Flow Analysis Algorithm", Intl. J. Computer Math., Vol. 3, 1971.

[Kildall 73] G.A. Kildall, "A Unified Approach to Global Program Optimization", Proc. ACM Symp. Principles of Programming Languages, 1973.

[Knuth 68] D.E. Knuth, "Semantics of Context-free Languages", Math. Systems Theory, Vol. 2 No. 2 pp. 127-145, June 1968.

[Koster 74] C.H.A. Koster, "Using the CDL Compiler-Compiler", in Compiler Construction: An Advanced Course, F.L. Bauer and J. Eickel, eds., Springer-Verlag, pp. 366-426, Berlin 1974.

[Lesk 79] M.E. Lesk, "Lex - A Lexical Analyzer Generator", UNIX Programmer's Manual 2, Section 20, 1979.

[Lewis 76] P.M. Lewis, II, D.J. Rosenkrantz and R.E. Stearns, Compiler Design Theory, Addison-Wesley, Reading, Mass., 1976.

[McKeeman 65]	W.M. McKeeman, "Peephole Optimization", CACM, Vol 8. No. 7, 1965.
[Miller 71]	P.L. Miller, "Automatic Creation of a Code Generator from a Machine Description", M.I.T. Tech Report MAC TR-85, 1971.
[Milton 77]	D.R. Milton, "Syntactic Specification and Analysis with Attribute Grammars", PhD thesis, University of Wisconsin-Madison, 1977.
[Milton 79]	D.R. Milton et al., "An ALL(1) Compiler Generator", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.
[Newcomer 75]	J.M. Newcomer, "Machine Independent Generation of Optimized Local Code", PhD thesis, Computer Science Dept., Carnegie Mellon University, 1975.
[Newell 69]	A. Newell and G.W. Ernst, "GPS: A Case Study in Generality and Problem Solving", Academic Press, 1969.
[Oakley 79]	J.D. Oakley, "Symbolic Execution of Formal Machine Descriptions", PhD thesis, Computer Science Dept., Carnegie-Mellon University, 1979.
[Raiha 8Ø]	K.J. Raiha, "Bibliography on Attribute Grammars", ACM Sigplan Notices, Vol. 15 No. 3 pp. 35-44, Mar 198Ø.
[Richards 71]	M. Richards, "The Portability of the BCPL Compiler", Software Practice and Experience, 1, pp. 135-146, 1971.
[Ripken 77]	K. Ripken, "Formale Beschreibun von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe", Technische Univer. Munchen, Munich, Germany, July 1977.
[Ritchie 78]	D.M. Ritchie and B.W. Kernighan, "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Robertson 77] E.L. Robertson, "Code Generation for Short/Long Address Machines", Tech. Report, Computer Sciences Dept., University of Wisconsin-Madison, 1977.

[Sethi 70] R. Sethi and J.D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions", JACM Vol. 17 No. 4, 1970.

[Skedzeleski78] S.K. Skedzeleski, "Definition and Use of Attribute Reevaluation in Attribute Grammars", PhD thesis, University of Wisconsin-Madison, 1978.

[Snyder 75] A. Snyder, "A Portable Compiler for the Language C", master's thesis, MIT, Cambridge, Mass., May 1975.

[Standish 76] T.A. Standish, D.C. Harriman, D.F. Kibler and J.M. Neighbors, "The Irvine Program Transformation Catalogue", University of California, Irvine, Dept. of Information and Computer Science, 1976.

[Steel 61] T.B. Steel, Jr., "A First Version of UNCOL", Proceedings WJCC, 19, pp. 371-378, 1961.

[Strong 58] J. Strong et al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution", CACM Vol.1 No. 8 pp. 12-18, 1958.

[Szymanski 78] T.G. Szymanski, "Assembling Code for Machines with Span-Dependent Instructions", CACM, Vol. 21 No. 4 pp. 300-308, April 1978.

[Szymanski 80] T.G. Szymanski and B. Leverett, "Chaining Span-Dependent Jump Instructions", ACM Transactions on Programming Languages and Systems, Vol. 2 No. 3, 1980.

[Ullman 75] J.D. Ullman, "Data Flow Analysis", Proc. 2nd USA-Japan Computer Conf., AFIPS Press, Montvale, N.J., 1975.

[Watt 74] D.A. Watt, "L.R. Parsing of Affix Grammars", PhD thesis, University of Glasgow, Report #7, 1974.

[Watt 77]	D.A. Watt, "The Parsing Problem for Affix Grammars", Acta Informatica, Springer Verlag, 1977.
[Weingart 73]	S.W. Weingart, "An Efficient and Systematic Method of Compiler Code Generation", PhD thesis, Computer Sciences Dept., Yale University, 1973.
[Wick 75]	J.D. Wick, "Automatic Generation of Assemblers", PhD Dissertation, Yale University, 1975.
[Wilcox 71]	T.R. Wilcox, "Generating Machine Code for High Level Programming Languages", Tech. Report 71-1Ø3, PhD thesis, Dept. of Computer Sciences, Cornell University, 1971.
[Wulf 75]	W. Wulf et al. "The Design of an Optimizing Compiler", American Elsevier Publishing Co., 1975.
[Wulf 79]	W. Wulf et al., "An Overview of the Production Quality Compiler-Compiler Project", Tech. Report CMU-CS-79-1Ø5, Carnegie Mellon University, Feb. 1979.
[Wulf 8Øa]	W. Wulf et al., "TCOL_{Ada}: Revised Report on An Intermediate Representation for the Preliminary Ada language", Tech. Report CMU-CS-8Ø-1Ø5, Dept. of Computer Science, Carnegie-Mellon University, Feb. 198Ø.
[Wulf 8Øb]	W. Wulf et al., "An Overview of the Production-Quality Compiler-Compiler Project", IEEE Computer Vol. 13 No. 8 pp. 38-49, Aug. 198Ø.
[Young 74]	R. Young, "The Coder: A Program Module for Code Generation in High Level Language Compilers", M.S. thesis, Computer Sciences Dept., University of Illinois, 1974.

Appendix A: Intermediate Representation

The intermediate representation is composed of operators, operands and attributes. Attributes are associated with operators as well as operands. For succinctness, we have taken some liberties with the usual attribute grammar formalism. Variable numbers of attributes may occur and their order of occurrence is not important. The obvious domain rules apply (e.g. the variables global, local, display have SCOPE as their domain, and character, integer, real have TYPE as their domain). Each operator is given with its arity and attributes.

Attribute Domains	Attribute Values
Amount	positive integer
Branch	branch op-codes
Kind	procedure, function
Optkind	time, space
Order	obverse, reverse
Prefer	register
Return	character, integer, long integer, pointer, real
Scope	global, static, local, chain, display, external
Type	character, integer, long integer, pointer, real

Operands	Attribute Domains
SIZE (size of data type)	Type
Real number	
Unsigned integer	
Identifier	Amount, Prefer, Scope, Type
Identifier (procedure/function name)	Kind, Order, Return

Operator	arity	Attributes	Meaning
:	1		beginning of declaration
{	Ø		opening of new scope
;	Ø		end of variable declaration
:=	2	Optkind	assignment
@	1	Type	indirection
,	2		indexing
#	1		address of variable
goto	1		unconditional branch
}	Ø		end of current scope
call	1		procedure or function call
<	3	Branch	less than
>	3	Branch	greater than
=	3	Branch	equals
<=	3	Branch	less than or equal to
>=	3	Branch	greater than or equal to
<>	3	Branch	not equal to
Ø<	2	Branch	greater than Ø
Ø>	2	Branch	less than Ø
Ø=	2	Branch	equals Ø
Ø<=	2	Branch	greater than or equal to Ø
Ø>=	2	Branch	less than or equal to Ø
Ø<>	2	Branch	not equal to Ø
+	2	Optkind	addition
-	2	Optkind	subtraction
*	2	Optkind	multiplication
/	2	Optkind	division
&	2	Optkind	boolean and
|	2	Optkind	boolean or
!	2	Optkind	exclusive or
~	1	Optkind	boolean not
~-	1	Optkind	unary negate
~&	2	Optkind	bitwise boolean and
~|	2	Optkind	bitwise boolean or
~!	2	Optkind	bitwise exclusive or
~~	1	Optkind	bitwise boolean not

Appendix B: Addressing-Mode Tables

Bits	Meaning
Ø (rightmost) and 1	Ø, 1, 2 or 3 levels of indirection
2	1 if base register is used
3	1 if displacement field is used
4	1 if index register field is used
5 and 6	ØØ auto-decrement
	Ø1 auto-increment
	1Ø and 11 not used

To represent additional addressing-mode properties more bits are required (e.g. to represent addressing modes of architectures that support more than three levels of indirection). Entries are ordered according to the numerical value of their bit specification. Some bit combinations are not supported by the architecture (they are marked "??"). Along with each entry, a factor (in bytes) is used to indicate the addressing mode's contribution towards instruction size. "%d" represents an integer value and "%s" represents a symbol string.

VAX-11/78Ø combination	format	size factor
ØØØØØØØ	??	Ø
ØØØØØØ1	??	Ø
ØØØØ1ØØ	??	Ø
ØØØØ1Ø1	(r%d)+	1
ØØØØ11Ø	*(r%d)+	1
ØØØ1ØØ1	??	Ø
ØØØ11ØØ	??	Ø
ØØØ11Ø1	??	Ø
ØØ1ØØØØ	??	Ø
ØØ1ØØØ1	??	Ø
ØØ1Ø1ØØ	??	Ø
ØØ1Ø1Ø1	(r%d)+[r%d]	2
ØØ1Ø11Ø	*(r%d)+[r%d]	2
ØØ11ØØ1	??	Ø
ØØ111ØØ	??	Ø

ØØ111Ø1	??	Ø
Ø1ØØØØØ	??	Ø
Ø1ØØØØ1	??	Ø
Ø1ØØ1ØØ	??	Ø
Ø1ØØ1Ø1	-(r%d)	1
Ø1Ø1ØØØ	??	Ø
Ø1Ø1ØØ1	??	Ø
Ø1Ø11ØØ	??	Ø
Ø1Ø11Ø1	??	Ø
Ø11ØØØØ	??	Ø
Ø11ØØØ1	??	Ø
Ø11Ø1ØØ	??	Ø
Ø11Ø1Ø1	-(r%d)[r%d]	2
Ø11Ø11Ø	??	Ø
Ø111ØØ1	??	Ø
Ø1111ØØ	??	Ø
Ø1111Ø1	??	Ø
1ØØØØØØ	??	Ø
1ØØØØØ1	??	Ø
1ØØØ1ØØ	r%d	1
1ØØØ1Ø1	(r%d)	1
1ØØ1ØØØ	$%d	2
1ØØ1ØØ1	*$%d	2
1ØØ11ØØ	%d(r%d)	2
1ØØ11Ø1	*%d(r%d)	2
1Ø1ØØØØ	??	Ø
1Ø1ØØØ1	??	Ø
1Ø1Ø1ØØ	??	Ø
1Ø1Ø1Ø1	(r%d)[r%d]	2
1Ø11ØØØ	$%d[r%d]	3
1Ø11ØØ1	*$%d[r%d]	3
1Ø111ØØ	%d(r%d)[r%d]	3
1Ø111Ø1	*%d(r%d)[r%d]	3
11ØØØØØ	%s	2
11ØØØØ1	*%s	2
11ØØ1ØØ	%s(r%d)	2
11ØØ1Ø1	*%s(r%d)	2
11Ø1ØØØ	%s+%d	2
11Ø1ØØ1	*%s+%d	2
11Ø11ØØ	??	2
11Ø11Ø1	??	2
111ØØØØ	%s[r%d]	3
111ØØØ1	*%s[r%d]	3
111Ø1ØØ	??	Ø
1111ØØØ	%s+%d[r%d]	3
1111ØØ1	*%s+%d[r%d]	3
11111ØØ	??	Ø
11111Ø1	??	Ø

Appendix C: Op-code Tables

```
#define CONDITION_CODE  4       /* 4 condition codes */
```

A = arithmetic operations and condition code setting
B = condition code set by both operands
C = op-code used only for condition-code setting
N = condition code not affected
R = condition code set by any operand
T = condition code only tested
Ø = condition code set to zero
1 = condition code set to one
Instruction timings are not available from manufacturer.

VAX	opcode	size	n	z	v	c	who sets cc	use
	addb2	1	R	R	R	R	2	A
	addb3	1	R	R	R	R	3	A
	addd2	1	R	R	R	Ø	2	A
	addd3	1	R	R	R	Ø	3	A
	addf2	1	R	R	R	Ø	2	A
	addf3	1	R	R	R	Ø	3	A
	addl2	1	R	R	R	R	2	A
	addl3	1	R	R	R	R	3	A
	addw2	1	R	R	R	R	2	A
	addw3	1	R	R	R	R	3	A
	aobleq	1	R	R	R	N	2	A
	aoblss	1	R	R	R	N	2	A
	ashl	1	R	R	R	Ø	3	A
	beql	1	N	T	N	N	Ø	C
	bgeq	1	T	N	N	N	Ø	C
	bgtr	1	T	T	N	N	Ø	C
	bisb2	1	R	R	Ø	N	2	A
	bisb3	1	R	R	Ø	N	3	A
	bisl2	1	R	R	Ø	N	2	A
	bisl3	1	R	R	Ø	N	3	A
	bisw2	1	R	R	Ø	N	2	A
	bisw3	1	R	R	Ø	N	3	A
	bitb	1	R	R	Ø	N	Ø	C
	bitl	1	R	R	Ø	N	Ø	C
	bitw	1	R	R	Ø	N	Ø	C
	bleq	1	T	T	N	N	Ø	C
	blss	1	T	N	N	N	Ø	C
	bneq	1	N	T	N	N	Ø	C
	brb	2	N	N	N	N	Ø	C
	brw	3	N	N	N	N	Ø	C
	calls	1	Ø	Ø	Ø	Ø	Ø	A
	clrb	1	Ø	1	Ø	N	1	A
	clrd	1	Ø	1	Ø	N	1	A
	clrf	1	Ø	1	Ø	N	1	A
	clrl	1	Ø	1	Ø	N	1	A

clrq	1	Ø	1	Ø	N	1	A
clrw	1	Ø	1	Ø	N	1	A
cmpb	1	B	B	Ø	B	Ø	C
cmpd	1	B	B	Ø	B	Ø	C
cmpf	1	B	B	Ø	B	Ø	C
cmpl	1	B	B	Ø	B	Ø	C
cmpw	1	B	B	Ø	B	Ø	C
cvtbd	1	R	R	R	N	2	A
cvtbf	1	R	R	R	N	2	A
cvtbl	1	R	R	R	N	2	A
cvtbw	1	R	R	R	N	2	A
cvtdb	1	R	R	R	N	2	A
cvtdf	1	R	R	R	N	2	A
cvtdl	1	R	R	R	N	2	A
cvtdw	1	R	R	R	N	2	A
cvtfb	1	R	R	R	N	2	A
cvtfd	1	R	R	R	N	2	A
cvtfl	1	R	R	R	N	2	A
cvtfw	1	R	R	R	N	2	A
cvtlb	1	R	R	R	N	2	A
cvtld	1	R	R	R	N	2	A
cvtlf	1	R	R	R	N	2	A
cvtlw	1	R	R	R	N	2	A
cvtrdl	1	R	R	R	N	2	A
cvtrfl	1	R	R	R	N	2	A
cvtwb	1	R	R	R	N	2	A
cvtwd	1	R	R	R	N	2	A
cvtwf	1	R	R	R	N	2	A
cvtwl	1	R	R	R	N	2	A
decb	1	R	R	R	R	1	A
decl	1	R	R	R	R	1	A
decw	1	R	R	R	R	1	A
divb2	1	R	R	R	N	2	A
divb3	1	R	R	R	N	3	A
divd2	1	R	R	R	N	2	A
divd3	1	R	R	R	N	3	A
divf2	1	R	R	R	N	2	A
divf3	1	R	R	R	N	3	A
divl2	1	R	R	R	N	2	A
divl3	1	R	R	R	N	3	A
divw2	1	R	R	R	N	2	A
divw3	1	R	R	R	N	3	A
incb	1	R	R	R	R	1	A
incl	1	R	R	R	R	1	A
incw	1	R	R	R	R	1	A
jmp	5	N	N	N	N	Ø	A
mcomb	1	R	R	Ø	N	2	A
mcoml	1	R	R	Ø	N	2	A
mcomw	1	R	R	Ø	N	2	A
mnegb	1	R	R	R	R	2	A

mnegd	1	R	R	Ø	Ø	2	A
mnegf	1	R	R	Ø	Ø	2	A
mnegl	1	R	R	R	R	2	A
mnegw	1	R	R	R	R	2	A
movab	1	R	R	Ø	N	2	A
movad	1	R	R	Ø	N	2	A
movaf	1	R	R	Ø	N	2	A
moval	1	R	R	Ø	N	2	A
movaq	1	R	R	Ø	N	2	A
movaw	1	R	R	Ø	N	2	A
movb	1	R	R	Ø	N	2	A
movd	1	R	R	Ø	N	2	A
movf	1	R	R	Ø	N	2	A
movl	1	R	R	Ø	N	2	A
movq	1	R	R	Ø	N	2	A
movw	1	R	R	Ø	N	2	A
mulb2	1	R	R	R	N	2	A
mulb3	1	R	R	R	N	3	A
muld2	1	R	R	R	N	2	A
muld3	1	R	R	R	N	3	A
mulf2	1	R	R	R	N	2	A
mulf3	1	R	R	R	N	3	A
mull2	1	R	R	R	N	2	A
mull3	1	R	R	R	N	3	A
mulw2	1	R	R	R	N	2	A
mulw3	1	R	R	R	N	3	A
pushab	1	R	R	Ø	N	1	A
pushad	1	R	R	Ø	N	1	A
pushaf	1	R	R	Ø	N	1	A
pushal	1	R	R	Ø	N	1	A
pushaq	1	R	R	Ø	N	1	A
pushaw	1	R	R	Ø	N	1	A
pushl	1	R	R	Ø	N	1	A
sobgeq	1	R	R	R	N	1	A
sobgtr	1	R	R	R	N	1	A
subb2	1	R	R	R	R	2	A
subb3	1	R	R	R	N	3	A
subd2	1	R	R	R	N	2	A
subd3	1	R	R	R	N	3	A
subf2	1	R	R	R	N	2	A
subf3	1	R	R	R	N	3	A
subl2	1	R	R	R	R	2	A
subl3	1	R	R	R	N	3	A
subw2	1	R	R	R	R	2	A
subw3	1	R	R	R	N	3	A
tstb	1	R	R	Ø	Ø	1	C
tstd	1	R	R	Ø	Ø	1	C
tstf	1	R	R	Ø	Ø	1	C
tstl	1	R	R	Ø	Ø	1	C
tstw	1	R	R	Ø	Ø	1	C

Appendix D: VAX/780 Attribute Grammar

Addressing mode productions

```
Address↑a → DirectModes↑a
        → IndirectModes↑a
IndirectModes↑a → @ DirectModes↑b NotIndirect (↓b)
                                ADDR (↓@ ↓b ↑a)
        → AnotherLevel↑a
DirectModes↑a → Datum↑a
        → # Datum↑b NotAssign
                                ADDR (↓# ↓b ↑a)
        → , Disp↑b Base↑c       ADDR (↓b ↓c ↑a)
        → , , Disp↑b Base↑c Index↑d
                                ADDR (↓b↓c↓d↑a)
        → Register
        → Subsumptions
Base↑a  → DirectModes↑a IsReg (↓a)
        → DirectModes↑b         GETREG (↓'long' ↑a)
                                EMIT (↓'movl' ↓b ↓a)
        → IndirectModes↑a IsReg (↓a)
        → IndirectModes↑b       GETREG (↓'long' ↑a)
                                EMIT (↓'movl' ↓b ↓a)
AnotherLevel↑a → @ IndirectModes↑b
                                GETREG (↓'long' ↑r)
                                EMIT (↓'movl' ↓b ↓r)
                                ADDR (↓@ ↓r ↑a)
Subsumptions↑a → @ + Byte↑b Byte↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓+↓b↓c↑a)
        → @ + Byte↑b Byte↑c Iscons (↓b) IsReg (↓c)
                                ADDR (↓+↓b↓c↑a)
        → @ + Word↑b Word↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓+↓b↓c↑a)
        → @ + Word↑b Word↑c Iscons (↓b) IsReg (↓c)
                                ADDR (↓+↓b↓c↑a)
        → @ + Long↑b Long↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓+↓b↓c↑a)
        → @ + Long↑b Long↑c Iscons (↓b) IsReg (↓c)
                                ADDR (↓+↓b↓c↑a)
        → @ - Byte↑b Byte↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓-↓b↓c↑a)
        → @ - Word↑b Word↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓-↓b↓c↑a)
        → @ - Long↑b Long↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓-↓b↓c↑a)
```

Instruction selection productions

```
Byte↑a   → Address↑a IsByte (↓a)
Word↑a   → Address↑a IsWord (↓a)
Long↑a   → Address↑a IsLong (↓a)
Float↑a  → Address↑a IsFloat (↓a)
Double↑a → Address↑a IsDouble (↓a)
Quad↑a   → Address↑a IsQuad (↓a)
```

Data transfer instructions

```
Assignment → := Byte↑a Byte↑b IsZero (↓b) EMIT (↓'clrb'↓a)
           → := Byte↑a Byte↑b     DELAY (↓'movb'↓b↓a)
           → := Word↑a Word↑b IsZero (↓b) EMIT (↓'clrw'↓a)
           → := Word↑a Word↑b     DELAY (↓'movw'↓b↓a)
           → := Long↑a Long↑b IsStack (↓a) EMIT (↓'pushl'↓b)
           → := Long↑a Long↑b IsZero (↓b) EMIT (↓'clrl'↓a)
           → := Long↑a Long↑b     DELAY (↓'movl'↓b↓a)
           → := Quad↑a Quad↑b IsZero (↓b) EMIT (↓'clrq'↓a)
           → := Quad↑a Quad↑b     DELAY (↓'movq'↓b↓a)
           → := Float↑a Float↑b IsZero (↓b) EMIT (↓'clrf'↓a)
           → := Float↑a Float↑b   DELAY (↓'movf'↓b↓a)
           → := Double↑a Double↑b IsZero (↓b) EMIT (↓'clrd'↓a)
           → := Double↑a Double↑b DELAY (↓'movd'↓b↓a)
           → := Long↑a #Byte↑b IsStack (↓a) EMIT (↓'pushab'↓b)
           → := Long↑a # Byte↑b   EMIT (↓'movab'↓b↓a)
           → := Long↑a #Word↑b IsStack (↓a) EMIT (↓'pushaw'↓b)
           → := Long↑a # Word↑b   EMIT (↓'movaw'↓b↓a)
           → := Long↑a #Long↑b IsStack (↓a) EMIT (↓'pushal'↓b)
           → := Long↑a # Long↑b   EMIT (↓'moval'↓b↓a)
           → := Long↑a #Float↑b IsStack (↓a) EMIT(↓'pushaf'↓b)
           → := Long↑a # Float↑b  EMIT (↓'movaf'↓b↓a)
           → := Long↑a #Quad↑b IsStack (↓a) EMIT(↓'pushaq'↓b)
           → := Long↑a # Quad↑b   EMIT (↓'movaq'↓b↓a)
           → := Long↑a # Double↑b IsStack (↓a)
                                  EMIT(↓'pushad'↓b)
           → := Long↑a # Double↑b EMIT (↓'movad'↓b↓a)
```

Special instructions

```
Special → := Long↑d - Long↑a Long↑b 0<= Long↑c Label↑n
                      IsOne (↓b) SobOk (↓d↓a↓c↓n)
                                 EMIT (↓'sobgeq'↓d↓n)
        → := Long↑d - Long↑a Long↑b 0< Long↑c Label↑n
                      IsOne (↓b) SobOk (↓d↓a↓c↓n)
                                 EMIT (↓'sobgtr'↓d↓n)
```

```
→ := Long↑d - Long↑a Long↑b
        Ørelop↑br Long↑c Label↑n IsOne (↓b)
                            AUTODEC (↓'decl'↓d)
                            EMIT (↓'tstl'↓c)
                            EMIT (↓br ↓n)
→ := Long↑d + Long↑a Long↑b
        <= Long↑e Long↑f Label↑n IsOne (↓b)
        AobOk (↓d↓a↓e↓n) EMIT (↓'aobleq'↓f↓d↓n)
→ := Long↑d + Long↑a Long↑b
        <= Long↑e Long↑f Label↑n IsOne (↓a)
        AobOk (↓d↓b↓e↓n) EMIT (↓'aobleq'↓f↓d↓n)
→ := Long↑d + Long↑a Long↑b
        >= Long↑e Long↑f Label↑n IsOne (↓b)
        AobOk (↓d↓a↓f↓n) EMIT (↓'aoblss'↓e↓d↓n)
→ := Long↑d + Long↑a Long↑b
        >= Long↑e Long↑f Label↑n IsOne (↓a)
        AobOk (↓d↓b↓f↓n) EMIT (↓'aoblss'↓e↓d↓n)
→ := Long↑d + Long↑a Long↑b
        < Long↑e Long↑f Label↑n IsOne (↓b)
        AobOk (↓d↓a↓e↓n) EMIT (↓'aoblss'↓f↓d↓n)
→ := Long↑d + Long↑a Long↑b
        < Long↑e Long↑f Label↑n IsOne (↓a)
        AobOk (↓d↓b↓e↓n) EMIT (↓'aoblss'↓f↓d↓n)
→ := Long↑d + Long↑a Long↑b
        > Long↑e Long↑f Label↑n IsOne (↓b)
        AobOk (↓d↓a↓f↓n) EMIT (↓'aobleq'↓e↓d↓n)
→ := Long↑d + Long↑a Long↑b
        > Long↑e Long↑f Label↑n IsOne (↓a)
        AobOk (↓d↓b↓f↓n) EMIT (↓'aobleq'↓e↓d↓n)
→ Long↑d + Long↑a Long↑b Ørelop↑br Long↑c Label↑n
                IsOne (↓b)
                            AUTOINC (↓'incl'↓d)
                            EMIT (↓'tstl'↓c)
                            EMIT (↓br ↓n)
→ Long↑d + Long↑a Long↑b Ørelop↑br Long↑c Label↑n
                IsOne (↓a)
                            AUTOINC (↓'incl'↓d)
                            EMIT (↓'tstl'↓c)
                            EMIT (↓br ↓n)
→ Long↑d + Long↑a Long↑b
        relop↑br Long↑e Long↑f Label↑n IsOne (↓b)
                            AUTOINC (↓'incl'↓d)
                            EMIT (↓'cmpl'↓e↓f)
                            EMIT(↓br ↓n)
→ Long↑d + Long↑a Long↑b
        relop↑br Long↑e Long↑f Label↑n IsOne (↓a)
                            AUTOINC (↓'incl'↓d)
                            EMIT (↓'cmpl'↓e↓f)
                            EMIT(↓br ↓n)
```

Arithmetic and Boolean instructions

```
Byte↑r  →  ~- Byte↑a GETTEMP (↓'byte' ↑r)
                            EMIT (↓'mnegb'↓a↓r)
        →  ~ Byte↑a GETTEMP (↓'byte' ↑r) EMIT (↓'mcomb'↓a↓r)
        →  + Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓+↓a↓b↑r)
        →  + Byte↑a Byte↑r IsOne (↓a) IsTemp (↓r)
                            AUTOINC (↓'incb'↓r)
        →  + Byte↑r Byte↑a IsOne (↓a) IsTemp (↓r)
                            AUTOINC (↓'incb'↓r)
        →  + Byte↑a Byte↑r TwoFourEight (↓a) IsTemp (↓r)
                            AUTOINC (↓'addb2'↓a↓r)
        →  + Byte↑r Byte↑a TwoFourEight (↓a) IsTemp (↓r)
                            AUTOINC (↓'addb2'↓a↓r)
        →  + Byte↑a Byte↑r TwoOp (↓+↓a↓r)
                            EMIT (↓'addb2'↓a↓r)
        →  + Byte↑r Byte↑a TwoOp (↓+↓a↓r)
                            EMIT (↓'addb2'↓a↓r)
        →  + Byte↑a Byte↑b GETTEMP (↓'byte' ↑r)
                            EMIT (↓'addb3'↓a↓b↓r)
        →  * Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓*↓a↓b↑r)
        →  * Byte↑a Byte↑r TwoOp (↓*↓a↓r)
                            EMIT (↓'mulb2'↓a↓r)
        →  * Byte↑r Byte↑a TwoOp (↓*↓a↓r)
                            EMIT (↓'mulb2'↓a↓r)
        →  * Byte↑a Byte↑b GETTEMP (↓'byte' ↑r)
                            EMIT (↓'mulb3'↓a↓b↓r)
        →  - Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓-↓a↓b↑r)
        →  - Byte↑r Byte↑a IsOne (↓a) IsTemp (↓r)
                            AUTODEC (↓'decb'↓r)
        →  - Byte↑r Byte↑a TwoFourEight (↓a) IsTemp (↓r)
                            AUTODEC (↓'subb2'↓a↓r)
        →  - Byte↑r Byte↑a TwoOp (↓-↓r↓a)
                            EMIT (↓'subb2'↓a↓r)
        →  - Byte↑a Byte↑b GETTEMP (↓'byte' ↑r)
                            EMIT (↓'subb3'↓a↓b↓r)
        →  / Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓/↓a↓b↑r)
        →  / Byte↑r Byte↑a IsTemp (↓r) EMIT (↓'divb2'↓a↓r)
        →  / Byte↑a Byte↑b GETTEMP (↓'byte' ↑r)
                            EMIT (↓'divb3'↓a↓b↓r)
        →  | Byte↑a Byte↑r TwoOp (↓|↓a↓r)
                            EMIT (↓'bisb2'↓a↓r)
        →  | Byte↑r Byte↑a TwoOp (↓|↓a↓r)
                            EMIT (↓'bisb2'↓a↓r)
        →  | Byte↑a Byte↑b GETTEMP (↓'byte' ↑r)
                            EMIT (↓'bisb3'↓a↓b↓r)
```

```
Word↑r  → ~- Word↑a GETTEMP (↓'word' ↑r)
                                EMIT (↓'mnegw'↓a↓r)
        → ~ Word↑a GETTEMP (↓'word' ↑r) EMIT (↓'mcomw'↓a↓r)
        → + Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓+↓a↓b↑r)
        → + Word↑a Word↑r IsOne (↓a) IsTemp (↓r)
                                AUTOINC (↓'incw'↓r)
        → + Word↑r Word↑a IsOne (↓a) IsTemp (↓r)
                                AUTOINC (↓'incw'↓r)
        → + Word↑a Word↑r TwoFourEight (↓a) IsTemp (↓r)
                                AUTOINC (↓'addw2'↓a↓r)
        → + Word↑r Word↑a TwoFourEight (↓a) IsTemp (↓r)
                                AUTOINC (↓'addw2'↓a↓r)
        → + Word↑a Word↑r TwoOp (↓+↓a↓r)
                                EMIT (↓'addw2'↓a↓r)
        → + Word↑r Word↑a TwoOp (↓+↓a↓r)
                                EMIT (↓'addw2'↓a↓r)
        → + Word↑a Word↑b GETTEMP (↓'word' ↑r)
                                EMIT (↓'addw3'↓a↓b↓r)
        → * Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓*↓a↓b↑r)
        → * Word↑a Word↑r TwoOp (↓*↓a↓r)
                                EMIT (↓'mulw2'↓a↓r)
        → * Word↑r Word↑a TwoOp (↓*↓a↓r)
                                EMIT (↓'mulw2'↓a↓r)
        → * Word↑a Word↑b GETTEMP (↓'word' ↑r)
                                EMIT (↓'mulw3'↓a↓b↓r)
        → - Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓-↓a↓b↑r)
        → - Word↑r Word↑a IsOne (↓a) IsTemp (↓r)
                                AUTODEC (↓'decw'↓r)
        → - Word↑r Word↑a TwoFourEight (↓a) IsTemp (↓r)
                                AUTODEC (↓'subw2'↓a↓r)
        → - Word↑r Word↑a TwoOp (↓-↓r↓a)
                                EMIT (↓'subw2'↓a↓r)
        → - Word↑a Word↑b GETTEMP (↓'word' ↑r)
                                EMIT (↓'subw3'↓a↓b↓r)
        → / Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓/↓a↓b↑r)
        → / Word↑r Word↑a IsTemp (↓r) EMIT (↓'divw2'↓a↓r)
        → / Word↑a Word↑b GETTEMP (↓'word' ↑r)
                                EMIT (↓'divw3'↓a↓b↓r)
        → | Word↑a Word↑r TwoOp (↓|↓a↓r)
                                EMIT (↓'bisw2'↓a↓r)
        → | Word↑r Word↑a TwoOp (↓|↓a↓r)
                                EMIT (↓'bisw2'↓a↓r)
        → | Word↑a Word↑b GETTEMP (↓'word' ↑r)
                                EMIT (↓'bisw3'↓a↓b↓r)
```

```
Long↑r  → ~- Long↑a GETTEMP (↓'long' ↑r)
                                    EMIT (↓'mnegl'↓a↓r)
        → ~ Long↑a GETTEMP (↓'long' ↑r) EMIT (↓'mcoml'↓a↓r)
        → + Long↑a Long↑b IsCons (↓a↓b) KFOLD (↓+↓a↓b↑r)
        → + Long↑a Long↑r IsOne (↓a) DontTryAob (↓a)
              IsTemp (↓r) AUTOINC (↓'incl'↓r)
        → + Long↑r Long↑a IsOne (↓a) DontTryAob (↓a)
              IsTemp (↓r) AUTOINC (↓'incl'↓r)
        → + Long↑a Long↑r TwoFourEight (↓a) IsTemp (↓r)
                                    AUTOINC (↓'addl2'↓a↓r)
        → + Long↑r Long↑a TwoFourEight (↓a) IsTemp (↓r)
                                    AUTOINC (↓'addl2'↓a↓r)
        → + Long↑a Long↑r TwoOp (↓+↓a↓r)
                                    EMIT (↓'addl2'↓a↓r)
        → + Long↑r Long↑a TwoOp (↓+↓a↓r)
                                    EMIT (↓'addl2'↓a↓r)
        → + Long↑a Long↑b GETTEMP (↓'long' ↑r)
                                    EMIT (↓'addl3'↓a↓b↓r)
        → * Long↑a Long↑b IsCons (↓a↓b) KFOLD (↓*↓a↓b↑r)
        → * Long↑a Long↑r IsTemp (↓r)
              TimeOpt (↓'ashl'↓'mull2') PowerTwo (↓a)
              LOG2 (↓a↑p) EMIT (↓'ashl'↓p↓r↓r)
        → * Long↑r Long↑a IsTemp (↓r)
              TimeOpt (↓'ashl'↓mull2') PowerTwo (↓a)
              LOG2 (↓a↑p) EMIT (↓'ashl'↓p↓r↓r)
        → * Long↑a Long↑r TwoOp (↓*↓a↓r)
                                    EMIT (↓'mull2'↓a↓r)
        → * Long↑r Long↑a TwoOp (↓*↓a↓r)
                                    EMIT (↓'mull2'↓a↓r)
        → * Long↑a Long↑b PowerTwo (↓a) LOG2 (↓a↑p)
              GETTEMP (↓'long' ↑r) EMIT (↓'ashl'↓p↓b↓r)
        → * Long↑a Long↑b PowerTwo (↓b) LOG2 (↓b↑p)
              GETTEMP (↓'long' ↑r) EMIT (↓'ashl'↓p↓a↓r)
        → * Long↑a Long↑b GETTEMP (↓'long' ↑r)
                                    EMIT (↓'mull3'↓a↓b↓r)
        → - Long↑a Long↑b IsCons (↓a↓b) KFOLD (↓-↓a↓b↑r)
        → - Long↑r  Long↑a  IsOne  (↓a)  DontTrySob  (↓a)
              IsTemp (↓r) AUTODEC (↓'decl'↓r)
        → - Long↑r Long↑a TwoFourEight (↓a) IsTemp (↓r)
                                    AUTODEC (↓'subl2'↓a↓r)
        → - Long↑r Long↑a TwoOp (↓-↓r↓a)
                                    EMIT (↓'subl2'↓a↓r)
        → - Long↑a Long↑b GETTEMP (↓'long' ↑r)
                                    EMIT (↓'subl3'↓a↓b↓r)
        → / Long↑a Long↑b IsCons (↓a↓b) KFOLD (↓/↓a↓b↑r)
        → / Long↑r Long↑a IsTemp (↓r) PowerTwo (↓a)
              MINUSLOG2 (↓a↑p) EMIT (↓'ashl'↓p↓r↓r)
        → / Long↑r Long↑a Istemp (↓r) EMIT (↓'divl2'↓a↓r)
        → / Long↑a Long↑b PowerTwo (↓b) MINUSLOG2 (↓b↑p)
              GETTEMP (↓'long' ↑r) EMIT (↓'ashl'↓p↓a↓r)
```

```
         → / Long↑a Long↑b GETTEMP (↓'long' ↑r)
                                   EMIT (↓'divl3'↓a↓b↓r)
         → | Long↑a Long↑r TwoOp (↓|↓a↓r)
                                   EMIT (↓'bisl2'↓a↓r)
         → | Long↑r Long↑a TwoOp (↓|↓a↓r)
                                   EMIT (↓'bisl2'↓a↓r)
         → | Long↑a Long↑b GETTEMP (↓'long' ↑r)
                                   EMIT (↓'bisl3'↓a↓b↓r)
Float↑r  → ~- Float↑a GETTEMP (↓'float' ↑r)
                                   EMIT (↓'mnegf'↓a↓r)
         → + Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓+↓a↓b↑r)
         → + Float↑a Float↑r TwoFourEight (↓a) IsTemp (↓r)
                                   AUTOINC (↓'addf2'↓a↓r)
         → + Float↑r Float↑a TwoFourEight (↓a) IsTemp (↓r)
                                   AUTOINC (↓'addf2'↓a↓r)
         → + Float↑a Float↑r TwoOp (↓+↓a↓r)
                                   EMIT (↓'addf2'↓a↓r)
         → + Float↑r Float↑a TwoOp (↓+↓a↓r)
                                   EMIT (↓'addf2'↓a↓r)
         → + Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                   EMIT (↓'addf3'↓a↓b↓r)
         → * Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓*↓a↓b↑r)
         → * Float↑a Float↑r TwoOp (↓*↓a↓r)
                                   EMIT (↓'mulf2'↓a↓r)
         → * Float↑r Float↑a TwoOp (↓*↓a↓r)
                                   EMIT (↓'mulf2'↓a↓r)
         → * Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                   EMIT (↓'mulf3'↓a↓b↓r)
         → - Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓-↓a↓b↑r)
         → - Float↑r Float↑a TwoFourEight (↓a) IsTemp (↓r)
                                   AUTODEC (↓'subf2'↓a↓r)
         → - Float↑r Float↑a TwoOp (↓-↓r↓a)
                                   EMIT (↓'subf2'↓a↓r)
         → - Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                   EMIT (↓'subf3'↓a↓b↓r)
         → / Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓/↓a↓b↑r)
         → / Float↑r Float↑a IsTemp (↓r) EMIT (↓'divf2'↓a↓r)
         → / Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                   EMIT (↓'divf3'↓a↓b↓r)
```

```
Double↑r→ ˜- Double↑a GETTEMP (↓'double' ↑r)
                                EMIT (↓'mnegd'↓a↓r)
         → + Double↑a Double↑b IsCons (↓a↓b)
                                KFOLD (↓+↓a↓b↑r)
         → + Double↑a Double↑r TwoFourEight (↓a) IsTemp (↓r)
                                AUTOINC (↓'addd2'↓a↓r)
         → + Double↑r Double↑a TwoFourEight (↓a) IsTemp (↓r)
                                AUTOINC (↓'addd2'↓a↓r)
         → + Double↑a Double↑r TwoOp (↓+↓a↓r)
                                EMIT (↓'addd2'↓a↓r)
         → + Double↑r Double↑a TwoOp (↓+↓a↓r)
                                EMIT (↓'addd2'↓a↓r)
         → + Double↑a Double↑b GETTEMP (↓'double' ↑r)
                                EMIT (↓'addd3'↓a↓b↓r)
         → * Double↑a Double↑b IsCons (↓a↓b)
                                KFOLD (↓*↓a↓b↑r)
         → * Double↑a Double↑r TwoOp (↓*↓a↓r)
                                EMIT (↓'muld2'↓a↓r)
         → * Double↑r Double↑a TwoOp (↓*↓a↓r)
                                EMIT (↓'muld2'↓a↓r)
         → * Double↑a Double↑b GETTEMP (↓'double' ↑r)
                                EMIT (↓'muld3'↓a↓b↓r)
         → - Double↑a Double↑b IsCons (↓a↓b)
                                KFOLD (↓-↓a↓b↑r)
         → - Double↑r Double↑a TwoFourEight (↓a)
                  IsTemp (↓r) AUTODEC (↓'subd2'↓a↓r)
         → - Double↑r Double↑a TwoOp (↓-↓r↓a)
                                EMIT (↓'subd2'↓a↓r)
         → - Double↑a Double↑b GETTEMP (↓'double' ↑r)
                                EMIT (↓'subd3'↓a↓b↓r)
         → / Double↑a Double↑b IsCons (↓a↓b)
                                KFOLD (↓/↓a↓b↑r)
         → / Double↑r Double↑a IsTemp (↓r)
                                EMIT(↓'divd2'↓a↓r)
         → / Double↑a Double↑b GETTEMP (↓'double' ↑r)
                                EMIT (↓'divd3'↓a↓b↓r)

/* Quadword specification similar to above data types */
```

Control instructions

```
Control → Cc↑br Label↑n EMIT (↓br↓n)
        → goto Label↑n  EMIT(↓'brb brw jmp' ↓n)
Cc↑br   → Ørelop↑br Byte↑a       EMIT (↓'tstb'↓a)
        → Ørelop↑br Word↑a       EMIT (↓'tstw'↓a)
        → Ørelop↑br Long↑a       EMIT (↓'tstl'↓a)
        → Ørelop↑br Float↑a      EMIT (↓'tstf'↓a)
        → Ørelop↑br Double↑a     EMIT (↓'tstd'↓a)
        → Relop↑br Byte↑a Byte↑b
                                 EMIT (↓'cmpb'↓a↓b)
        → Relop↑br Word↑a Word↑b
                                 EMIT (↓'cmpw'↓a↓b)
        → Relop↑br Long↑a Long↑b
                                 EMIT (↓'cmpl'↓a↓b)
        → Relop↑br Float↑a Float↑b
                                 EMIT (↓'cmpf'↓a↓b)
        → Relop↑br Double↑a Double↑b
                                 EMIT (↓'cmpd'↓a↓b)
        → And↑br Byte↑a Byte↑b   EMIT (↓'bitb'↓a↓b)
        → And↑br Word↑a Word↑b   EMIT (↓'bitw'↓a↓b)
        → And↑br Long↑a Long↑b   EMIT (↓'bitl'↓a↓b)
        → Or↑br Byte↑a Byte↑b    GETTEMP (↓'byte' ↑r)
                                 EMIT (↓'bisb3'↓a↓b↓r)
                                 FREETEMP (↓r)
        → Or↑br Word↑a Word↑b    GETTEMP (↓'word' ↑r)
                                 EMIT (↓'bisw3'↓a↓b↓r)
                                 FREETEMP (↓r)
        → Or↑br Long↑a Long↑b    GETTEMP (↓'long' ↑r)
                                 EMIT (↓'bisl3'↓a↓b↓r)
                                 FREETEMP (↓r)
 Ørelop↑'beql bneq' → Ø=
 Ørelop↑'bneq beql' → Ø<>
 Ørelop↑'blss bgeq' → Ø>
 Ørelop↑'bleq bgtr' → Ø>=
 Ørelop↑'bgtr bleq' → Ø<
 Ørelop↑'bgeq blss' → Ø<=
 Relop↑'beql bneq' → =
 Relop↑'bneq beql' → <>
 Relop↑'blss bgeq' → <
 Relop↑'bleq bgtr' → <=
 Relop↑'bgtr bleq' → >
 Relop↑'bgeq blss' → >=
 And↑'bneq beql'   → &
 Or↑'bneq beql'    → |
```

Procedure call instruction

```
Pcall → CALL Name↑a EMIT (↓'calls'↓a)
```

<u>Transfer productions</u>

```
BooleanByte↑r → ConvToByte GETTEMP (↓'byte' ↑r)
                                EMIT (↓'clrb' ↓r)
BooleanWord↑r → ConvToWord GETTEMP (↓'word' ↑r)
                                EMIT (↓'clrw' ↓r)
BooleanLong↑r → ConvToLong GETTEMP (↓'long' ↑r)
                                EMIT (↓'clrl' ↓r)
Byte↑r   → Word↑a ConvToByte (↓a) GETTEMP (↓'byte'↑r)
                                EMIT (↓'cvtwb'↓a↓r)
         → Long↑a ConvToByte (↓a) GETTEMP (↓'byte'↑r)
                                EMIT (↓'cvtlb'↓a↓r)
         → Float↑a ConvToByte (↓a) GETTEMP (↓'byte'↑r)
                                EMIT (↓'cvtfb'↓a↓r)
         → Double↑a ConvToByte (↓a) GETTEMP (↓'byte'↑r)
                                EMIT (↓'cvtdb'↓a↓r)
         → BooleanByte↑a Cc↑br  GETLAB (↑n)
                                EMIT (↓br ↓n)
                                EMIT (↓'incb' ↓a)
                                EMIT (↓n)
Word↑r   → Byte↑a ConvToWord (↓a) GETTEMP (↓'word'↑r)
                                EMIT (↓'cvtbw'↓a↓r)
         → Long↑a ConvToWord (↓a) GETTEMP (↓'word'↑r)
                                EMIT (↓'cvtlw'↓a↓r)
         → Float↑a ConvToWord (↓a) GETTEMP (↓'word'↑r)
                                EMIT (↓'cvtfw'↓a↓r)
         → Double↑a ConvToWord (↓a) GETTEMP (↓'word'↑r)
                                EMIT (↓'cvtdw'↓a↓r)
         → BooleanWord↑a Cc↑br  GETLAB (↑n)
                                EMIT (↓br ↓n)
                                EMIT (↓'incw' ↓a)
                                EMIT (↓n)
Long↑r   → Byte↑a ConvToLong (↓a) GETTEMP (↓'long'↑r)
                                EMIT (↓'cvtbl'↓a↓r)
         → Word↑a ConvToLong (↓a) GETTEMP (↓'long'↑r)
                                EMIT (↓'cvtwl'↓a↓r)
         → Float↑a ConvToLong (↓a) GETTEMP (↓'long'↑r)
                                EMIT (↓'cvtfl'↓a↓r)
         → Double↑a ConvToLong (↓a) GETTEMP (↓'long'↑r)
                                EMIT (↓'cvtdl'↓a↓r)
         → BooleanLong↑a Cc↑br  GETLAB (↑n)
                                EMIT (↓br ↓n)
                                EMIT (↓'incl' ↓a)
                                EMIT (↓n)
```

```
Float↑r → Word↑a ConvToFloat (↓a) GETTEMP (↓'float'↑r)
                                  EMIT (↓'cvtwf'↓a↓r)
        → Long↑a ConvToFloat (↓a) GETTEMP (↓'float'↑r)
                                  EMIT (↓'cvtlf'↓a↓r)
        → Byte↑a ConvToFloat (↓a) GETTEMP (↓'float'↑r)
                                  EMIT (↓'cvtbf'↓a↓r)
        → Double↑a ConvToFloat (↓a) GETTEMP (↓'float'↑r)
                                  EMIT (↓'cvtdf'↓a↓r)
Double↑r→ Word↑a ConvToDouble (↓a) GETTEMP (↓'double'↑r)
                                  EMIT (↓'cvtwd'↓a↓r)
        → Long↑a ConvToDouble (↓a) GETTEMP (↓'double'↑r)
                                  EMIT (↓'cvtld'↓a↓r)
        → Byte↑a ConvToDouble (↓a) GETTEMP (↓'double'↑r)
                                  EMIT (↓'cvtbd'↓a↓r)
        → Float↑a ConvToDouble (↓a) GETTEMP (↓'double'↑r)
                                  EMIT (↓'cvtfd'↓a↓r)
```

Action Symbol	function
ADDR	compose address attribute
AUTODEC	attempt auto-decrement optimization
AUTOINC	attempt auto-increment optimization
DELAY	delay assignment
EMIT	emit assembler code
FREETEMP	free attribute that is a temporary
GETLAB	obtain an internal label
GETREG	obtain a free register
GETTEMP	obtain a free temporary location
KFOLD	perform constant folding and then return address attribute
LOG2	return log2 of attribute's value
MINUSLOG2	return negative log2 of value

Predicate	Evaluates to true when
IsByte	data type of attribute is a byte
IsWord	data type of attribute is a word
IsLong	data type of attribute is a long
IsFloat	data type of attribute is a float
IsDouble	data type of attribute is a double
IsQuad	data type of attribute is a quad
IsStack	attribute is top of stack
IsZero	attribute is constant zero
IsOne	attribute is constant one
IsCons	attributes are constants
IsReg	attribute is a register location
IsTemp	attribute is a temporary
TwoFourEight	attribute is constant 2, 4, or 8
PowerTwo	attribute is a power of 2
TwoOp	Two-address op-code must be used
AobOk	Conditions for add-one-and-branch are satisfied
DontTrySob	Do not try for subtract-one-and-branch
SobOk	Conditions for subtract-one-and-branch are satisfied
TimeOpt	execution speed preferred over object-code size optimization
NotIndirect	attribute is not already an indirect addressing mode
NotAssign	address operator is not immediately used as RHS of assignment statement
ConvToByte	convert attribute to byte data type
ConvToWord	convert attribute to word data type
ConvToLong	convert attribute to long data type
ConvToFloat	convert attribute to float type
ConvToDouble	convert attribute to double type
ConvToQuad	convert attribute to quad data type

Appendix E: PDP-11/70 Attribute Grammar

Addressing mode productions

```
Address↑a → DirectModes↑a
        → IndirectModes↑a
IndirectModes↑a → @ DirectModes↑b NotIndirect (↓b)
                                ADDR (↓@ ↓b ↑a)
        → AnotherLevel↑a
DirectModes↑a → Datum↑a
        → # Datum↑b             ADDR (↓# ↓b ↑a)
        → , Disp↑b Base↑c       ADDR (↓b ↓c ↑a)
        → Register
        → Subsumptions
Base↑a  → DirectModes↑a IsReg (↓a)
        → DirectModes↑b GETREG (↓'word' ↑a)
                                EMIT (↓'mov' ↓b ↓a)
        → IndirectModes↑a IsReg (↓a)
        → IndirectModes↑b       GETREG (↓'word' ↑a)
                                EMIT (↓'mov' ↓b ↓a)
AnotherLevel↑a → @ IndirectModes↑b
                                GETREG (↓'word' ↑r)
                                EMIT (↓'mov' ↓b ↓r)
                                ADDR (↓@ ↓r ↑a)
Subsumptions↑a → @ + Byte↑b Byte↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓+↓b↓c↑a)
        → @ + Byte↑b Byte↑c Iscons (↓b) IsReg (↓c)
                                ADDR (↓+↓b↓c↑a)
        → @ + Word↑b Word↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓+↓b↓c↑a)
        → @ + Word↑b Word↑c Iscons (↓b) IsReg (↓c)
                                ADDR (↓+↓b↓c↑a)
        → @ - Byte↑b Byte↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓-↓b↓c↑a)
        → @ - Word↑b Word↑c Iscons (↓c) IsReg (↓b)
                                ADDR (↓-↓b↓c↑a)
```

Instruction selection productions

```
Byte↑a  → Address↑a IsByte (↓a)
Word↑a  → Address↑a IsWord (↓a)
Float↑a → Address↑a IsFloat (↓a)
Double↑a→ Address↑a IsDouble (↓a)
```

Data transfer instructions

```
Assignment → := Byte↑a Byte↑b IsZero (↓b) EMIT (↓'clrb'↓a)
        → := Byte↑a Byte↑b DELAY (↓'movb'↓b↓a)
        → := Word↑a Word↑b IsZero (↓b) EMIT (↓'clr'↓a)
        → := Word↑a Word↑b DELAY (↓'mov'↓b↓a)
        → := Float↑a Float↑b IsReg (↓b) DELAY (↓'stf'↓b↓a)
        → := Double↑a Double↑b IsReg (↓b) DELAY(↓'std'↓b↓a)
```

Special instructions

```
Special → := Word↑d - Word↑a Word↑b Ø<> Word↑c Label↑n
              IsOne (↓b) IsReg (↓d) SobOk (↓d↓a↓c↓n)
                                 EMIT (↓'sob'↓d↓n)
        → := Word↑d - Word↑a Word↑b
             Ørelop↑br Word↑c Label↑n IsOne (↓b)
                                 AUTODEC (↓'dec'↓d)
                                 EMIT (↓'tst'↓c)
                                 EMIT (↓br ↓n)
```

Arithmetic and Boolean instructions

```
Byte↑r  → ~- Byte↑r Istemp (↓r) EMIT (↓'negb' ↓r)
        → ~- Byte↑a GETTEMP (↓'byte' ↑r)
                                 EMIT (↓'mov' ↓a ↓r)
                                 EMIT (↓'negb' ↓r)
        → ~ Byte↑r Istemp (↓r) EMIT (↓'comb' ↓r)
        → ~ Byte↑a GETTEMP (↓'byte' ↑r) EMIT (↓'mov' ↓a ↓r)
                                 EMIT (↓'comb' ↓r)
        → + Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓+↓a↓b↑r)
        → + Byte↑a Byte↑r IsOne (↓a) IsTemp (↓r)
                                 AUTOINC (↓'incb'↓r)
        → + Byte↑r Byte↑a IsOne (↓a) IsTemp (↓r)
                                 AUTOINC (↓'incb'↓r)
        → * Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓*↓a↓b↑r)
        → * Byte↑a Byte↑r Two (↓a) IsTemp(↓r)
                                 EMIT (↓'aslb' ↓r)
        → * Byte↑r Byte↑a Two (↓a) IsTemp(↓r)
                                 EMIT (↓'aslb' ↓r)
        → - Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓-↓a↓b↑r)
        → - Byte↑r Byte↑a IsOne (↓a) IsTemp (↓r)
                                 AUTODEC (↓'decb'↓r)
        → / Byte↑a Byte↑b IsCons (↓a↓b) KFOLD (↓/↓a↓b↑r)
        → / Byte↑r Byte↑a IsTemp (↓r) Two (↓a)
                                 EMIT (↓'asrb'↓r)
        → | Byte↑a Byte↑r IsTemp (↓r) EMIT (↓'bisb'↓a↓r)
        → | Byte↑r Byte↑a IsTemp (↓r) EMIT (↓'bisb'↓a↓r)
        → | Byte↑a Byte↑b        GETTEMP (↓'byte' ↑r)
                                 EMIT (↓'movb'↓a↓r)
                                 EMIT (↓'bisb'↓b↓r)
```

```
Word↑r  → ~- Word↑r Istemp (↓r) EMIT (↓'neg' ↓r)
        → ~- Word↑a              GETTEMP (↓'word' ↑r)
                                 EMIT (↓'mov' ↓a ↓r)
                                 EMIT (↓'neg' ↓r)
        → ~ Word↑r Istemp (↓r) EMIT (↓'com' ↓r)
        → ~ Word↑a               GETTEMP (↓'word' ↑r)
                                 EMIT (↓'mov' ↓a ↓r)
                                 EMIT (↓'com' ↓r)
        → + Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓+↓a↓b↑r)
        → + Word↑a Word↑r IsOne (↓a) IsTemp (↓r)
                                 AUTOINC (↓'inc'↓r)
        → + Word↑r Word↑a IsOne (↓a) IsTemp (↓r)
                                 AUTOINC (↓'inc'↓r)
        → + Word↑a Word↑r TwoFour (↓a) IsTemp (↓r)
                                 AUTOINC (↓'add'↓a↓r)
        → + Word↑r Word↑a TwoFour (↓a) IsTemp (↓r)
                                 AUTOINC (↓'add'↓a↓r)
        → + Word↑a Word↑r IsTemp (↓r) EMIT (↓'add'↓a↓r)
        → + Word↑r Word↑a IsTemp (↓r) EMIT (↓'add'↓a↓r)
        → + Word↑a Word↑b GETTEMP (↓'word' ↑r)
                                 EMIT (↓'mov' ↓b ↓r)
                                 EMIT (↓'add' ↓a ↓r)
        → * Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓*↓a↓b↑r)
        → * Word↑a Word↑r Two (↓a) IsTemp (↓r)
                                 EMIT (↓'asl' ↓r)
        → * Word↑r Word↑a Two (↓a) IsTemp (↓r)
                                 EMIT (↓'asl' ↓r)
        → * Word↑r Word↑a PowerTwo (↓a) LOG2 (↓a ↑p)
                          IsTemp (↓r) EMIT (↓'ash' ↓p ↓r)
        → * Word↑a Word↑r PowerTwo (↓a) LOG2 (↓a ↑p)
                          IsTemp (↓r) EMIT (↓'ash' ↓p ↓r)
        → * Word↑a Word↑r IsTemp (↓r) EMIT (↓'mul'↓a↓r)
        → * Word↑r Word↑a IsTemp (↓r) EMIT (↓'mul'↓a↓r)
        → * Word↑a Word↑b        GETTEMP (↓'word' ↑r)
                                 EMIT (↓'mov' ↓b ↓r)
                                 EMIT (↓'mul' ↓a ↓r)
        → - Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓-↓a↓b↑r)
        → - Word↑r Word↑a IsOne (↓a) IsTemp (↓r)
                                 AUTODEC (↓'dec'↓r)
        → - Word↑r Word↑a TwoFour (↓a) IsTemp (↓r)
                                 AUTODEC (↓'sub'↓a↓r)
        → - Word↑r Word↑a IsTemp (↓r) EMIT (↓'sub'↓a↓r)
        → - Word↑a Word↑b        GETTEMP (↓'word' ↑r)
                                 EMIT (↓'mov'↓a↓r)
                                 EMIT (↓'sub'↓b↓r)
```

```
          → / Word↑a Word↑b IsCons (↓a↓b) KFOLD (↓/↓a↓b↑r)
          → / Word↑r Word↑a IsTemp (↓r) Two (↓a)
                                  EMIT (↓'asr'↓r)
          → / Word↑r Word↑a PowerTwo (↓a) MINUSLOG2 (↓a ↑p)
                IsTemp (↓r) EMIT (↓'ash' ↓p ↓r)
          → / Word↑r Word↑a IsTemp (↓r) EMIT (↓'div'↓a↓r)
          → / Word↑a Word↑b GETTEMP (↓'word' ↓'even' ↑r)
                                  EMIT (↓'mov'↓a↓r)
                                  EMIT (↓'div'↓b↓r)
          → | Word↑a Word↑r IsTemp (↓r) EMIT (↓'bis'↓a↓r)
          → | Word↑r Word↑a IsTemp (↓r) EMIT (↓'bis'↓a↓r)
          → | Word↑a Word↑b GETTEMP (↓'word' ↑r)
                                  EMIT (↓'mov'↓a↓r)
                                  EMIT (↓'bis'↓b↓r)
Float↑r → ~- Float↑r Istemp (↓r) EMIT (↓'negf' ↓r)
          → ~- Float↑a GETTEMP (↓'float' ↑r)
                                  EMIT (↓'ldf' ↓a ↓r)
                                  EMIT (↓'negf' ↓r)
          → + Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓+↓a↓b↑r)
          → + Float↑a Float↑r TwoFour (↓a) IsTemp (↓r)
                                  AUTOINC (↓'addf'↓a↓r)
          → + Float↑r Float↑a TwoFour (↓a) IsTemp (↓r)
                                  AUTOINC (↓'addf'↓a↓r)
          → + Float↑a Float↑r IsTemp (↓r) EMIT (↓'addf'↓a↓r)
          → + Float↑r Float↑a IsTemp (↓r) EMIT (↓'addf'↓a↓r)
          → + Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                  EMIT (↓'ldf' ↓b ↓r)
                                  EMIT (↓'addf' ↓a ↓r)
          → * Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓*↓a↓b↑r)
          → * Float↑a Float↑r IsTemp (↓r) EMIT (↓'mulf'↓a↓r)
          → * Float↑r Float↑a IsTemp (↓r) EMIT (↓'mulf'↓a↓r)
          → * Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                  EMIT (↓'ldf' ↓b ↓r)
                                  EMIT (↓'mulf' ↓b ↓r)
          → - Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓-↓a↓b↑r)
          → - Float↑r Float↑a TwoFour (↓a) IsTemp (↓r)
                                  AUTODEC (↓'subf'↓a↓r)
          → - Float↑r Float↑a IsTemp (↓r) EMIT (↓'subf'↓a↓r)
          → - Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                  EMIT (↓'ldf'↓a↓r)
                                  EMIT (↓'subf'↓b↓r)
          → / Float↑a Float↑b IsCons (↓a↓b) KFOLD (↓/↓a↓b↑r)
          → / Float↑r Float↑a IsTemp (↓r) EMIT (↓'divf'↓a↓r)
          → / Float↑a Float↑b GETTEMP (↓'float' ↑r)
                                  EMIT (↓'ldf'↓a↓r)
                                  EMIT (↓'divf'↓b↓r)
```

```
Double↑r→ ~- Double↑r Istemp (↓r) EMIT (↓'negd' ↓r)
        → ~- Double↑a GETTEMP (↓'double' ↑r)
                               EMIT (↓'ldd' ↓a ↓r)
                               EMIT (↓'negd' ↓r)
        → + Double↑a Double↑b IsCons (↓a↓b)
                               KFOLD (↓+↓a↓b↑r)
        → + Double↑a Double↑r TwoFour (↓a) IsTemp (↓r)
                               AUTOINC (↓'addd'↓a↓r)
        → + Double↑r Double↑a TwoFour (↓a) IsTemp (↓r)
                               AUTOINC (↓'addd'↓a↓r)
        → + Double↑a Double↑r IsTemp (↓r)
                               EMIT (↓'addd'↓a↓r)
        → + Double↑r Double↑a IsTemp (↓r)
                               EMIT (↓'addd'↓a↓r)
        → + Double↑a Double↑b GETTEMP (↓'double' ↑r)
                               EMIT (↓'ldd' ↓b ↓r)
                               EMIT (↓'addd' ↓a ↓r)
        → * Double↑a Double↑b IsCons (↓a↓b)
                               KFOLD (↓*↓a↓b↑r)
        → * Double↑a Double↑r IsTemp (↓r)
                               EMIT (↓'muld'↓a↓r)
        → * Double↑r Double↑a IsTemp (↓r)
                               EMIT (↓'muld'↓a↓r)
        → * Double↑a Double↑b GETTEMP (↓'double' ↑r)
                               EMIT (↓'ldd' ↓b ↓r)
                               EMIT (↓'muld' ↓ba ↓r)
        → - Double↑a Double↑b IsCons (↓a↓b)
                               KFOLD (↓-↓a↓b↑r)
        → - Double↑r Double↑a TwoFour (↓a) IsTemp (↓r)
                               AUTODEC (↓'subd'↓a↓r)
        → - Double↑r Double↑a IsTemp (↓r)
                               EMIT (↓'subd'↓a↓r)
        → - Double↑a Double↑b GETTEMP (↓'double' ↑r)
                               EMIT (↓'ldd'↓a↓r)
                               EMIT (↓'subd'↓b↓r)
        → / Double↑a Double↑b IsCons (↓a↓b)
                               KFOLD (↓/↓a↓b↑r)
        → / Double↑r Double↑a IsTemp (↓r)
                               EMIT (↓'divd'↓a↓r)
        → / Double↑a Double↑b GETTEMP (↓'double' ↑r)
                               EMIT (↓'ldd'↓a↓r)
                               EMIT (↓'divd'↓b↓r)
```

<u>Control instructions</u>

```
Control → Cc↑br Label↑n EMIT (↓br↓n)
        → goto Label↑n EMIT(↓'br jmp' ↓n)
Cc↑br   → Ørelop↑br Byte↑a EMIT (↓'tstb'↓a)
        → Ørelop↑br Word↑a EMIT (↓'tst'↓a)
        → Ørelop↑br Float↑a EMIT (↓'tstf'↓a)
        → Ørelop↑br Double↑a EMIT (↓'tstd'↓a)
        → Relop↑br Byte↑a Byte↑b EMIT (↓'cmpb'↓a↓b)
        → Relop↑br Word↑a Word↑b EMIT (↓'cmp'↓a↓b)
        → Relop↑br Float↑a Float↑b EMIT (↓'cmpf'↓a↓b)
        → Relop↑br Double↑a Double↑b EMIT (↓'cmpd'↓a↓b)
        → And↑br Byte↑a Byte↑b EMIT (↓'bitb'↓a↓b)
        → And↑br Word↑a Word↑b EMIT (↓'bit'↓a↓b)
        → Or↑br Byte↑a Byte↑b   GETTEMP (↓'byte' ↑r)
                                 EMIT (↓'movb' ↓b ↓r)
                                 EMIT (↓'bisb' ↓a ↓r)
        → Or↑br Word↑a Word↑b   GETTEMP (↓'word' ↑r)
                                 EMIT (↓'mov' ↓b ↓r)
                                 EMIT (↓'bis' ↓a ↓r)
Ørelop↑'beq bne' → Ø=
Ørelop↑'bne beq' → Ø<>
Ørelop↑'blt bge' → Ø>
Ørelop↑'ble bgt' → Ø>=
Ørelop↑'bgt ble' → Ø<
Ørelop↑'bge blt' → Ø<=
Relop↑'beq bne' → =
Relop↑'bne beq' → <>
Relop↑'blt bge' → <
Relop↑'ble bgt' → <=
Relop↑'bgt ble' → >
Relop↑'bge blt' → >=
And↑'bne beq' → &
```

<u>Procedure call instruction</u>

```
Pcall → CALL Name↑a EMIT (↓'jsr' ↓'FrameReg' ↓a)
```

Transfer productions

```
BooleanByte↑r → ConvToByte GETTEMP (↓'byte' ↑r)
                                EMIT (↓'clrb' ↓r)
BooleanWord ↑r → ConvToWord GETTEMP (↓'word' ↑r)
                                EMIT (↓'clr' ↓r)
Byte↑r   → Word↑a ConvToByte (↓a) GETTEMP (↓'byte'↑r)
                   conversion code sequence
         → Float↑a ConvToByte (↓a) GETTEMP (↓'byte'↑r)
                   conversion code sequence
         → Double↑a ConvToByte (↓a) GETTEMP (↓'byte'↑r)
                   conversion code sequence
         → BooleanByte↑a Cc↑br   GETLAB (↑n)
                                 EMIT (↓br ↓n)
                                 EMIT (↓'incb' ↓a)
                                 EMIT (↓n)
Word↑r   → Byte↑a ConvToWord (↓a) GETTEMP (↓'word'↑r)
                   conversion code sequence
         → Float↑a ConvToWord (↓a) GETTEMP (↓'word'↑r)
                   conversion code sequence
         → Double↑a ConvToWord (↓a) GETTEMP (↓'word'↑r)
                   conversion code sequence
         → BooleanWord↑a Cc↑br   GETLAB (↑n)
                                 EMIT (↓br ↓n)
                                 EMIT (↓'inc' ↓a)
                                 EMIT (↓n)
Float↑r → Word↑a ConvToFloat (↓a) GETTEMP (↓'float'↑r)
                   conversion code sequence
         → Byte↑a ConvToFloat (↓a) GETTEMP (↓'float'↑r)
                   conversion code sequence
         → Double↑a ConvToFloat (↓a) GETTEMP (↓'float'↑r)
                                 EMIT (↓ndcdf ↓a ↓r)
Double↑r→ Word↑a ConvToDouble (↓a) GETTEMP (↓'double'↑r)
                   conversion code sequence
         → Byte↑a ConvToDouble (↓a) GETTEMP (↓'double'↑r)
                   conversion code sequence
         → Float↑a ConvToDouble (↓a) GETTEMP (↓'double'↑r)
                                 EMIT (↓'ldcfd' ↓a ↓r)
```

Appendix F: Implementing Disambiguating Predicates in CFGs

Conflicts in bottom-up parsers (i.e. shift/reduce and reduce/reduce conflicts) are normally resolved by parser generators in favor of a fixed production (depending on look ahead). YACC [Johnson 75] uses a look-ahead symbol to resolve conflicts. If the look ahead does not suffice [Aho 75], shift/reduce conflicts are always resolved in favor of a shift, and reduce/reduce conflicts are always resolved in favor of the production that occurs lexically before the others that conflict. The user can therefore place the desired production before the others.

Often, as in this research, it becomes necessary to choose different productions from a conflicting set under different contexts. Disambiguating predicates are used to "dynamically" resolve conflicts. In a bottom-up parser, disambiguating predicates are (in principle) associated with every state and any configuration in the state. Upon the occurrence of a conflict, all relevant disambiguating predicates are evaluated, and the production for which the predicate evaluates to true is selected. The disambiguating predicates are evaluated in the order in which the productions are specified. Predicates are evaluated even if there is no conflict. The first production all of whose predicates evaluate to true is selected. This criterion ensures that at most one production is selected at any given time. In general, a hierarchy of disambiguating predicates can be used to select a production. In practice, a linear ordering seems to suffice.

The following discussion suggests a technique to incorporate disambiguating predicates within a context-free bottom-up parsing framework such as YACC by suitable modification of the parser-driver.

Consider the following productions:

[P_1] $a \rightarrow c$

[P_2] $b \rightarrow c$

[P_3] $d \rightarrow e\ a\ b$

[P_4] $d \rightarrow e\ b\ a$

A reduce/reduce conflict is caused by [P_1] and [P_2]. To disambiguate this conflict, we add a non-terminal 'V', a disambiguating routine 'disamb' and tokens T_a and T_b as follows:

[P_1] $a \rightarrow c\ V\ T_a$

[P_2] $b \rightarrow c\ V\ T_b$

[P_5] $V \rightarrow \epsilon$ disamb(T_a, T_b);

The decision to select [P_1] or [P_2] is made by disamb when a reduction by [P_5] occurs (i.e., P_5 "triggers" the disambiguating predicate). It looks at the context (or uses any conditions programmed by the user) and inserts either token T_a or T_b in the parser's input stream as an indication of its choice. The parser (if it uses look-ahead) may have already read in the look-ahead token. In this case, the disambiguating token must be inserted before any look-ahead token. If the token inserted by disamb is consumed before another similar insertion, then Buffer (explained in the next page) need only be a single global location. In general, for a k-token look-ahead parser, Buffer has to be a stack of depth k so that all k parser look-aheads can be pushed onto the buffer-stack before "disamb" inserts a token into the input stream. The following code illustrates this process.

```
PROCEDURE disamb(Token$_1$, .. ,Token$_n$)
    BEGIN
        IF predicate$_1$ evaluates to true THEN
            insert(Token$_1$)
        ELSE    IF predicate$_2$ evaluates to true THEN
                        insert(Token$_2$)
                ELSE    ...........
    END; (* procedure disamb *)

PROCEDURE insert(Token)
    BEGIN (* check if parser look-ahead exists *)
        IF parser look-ahead token exists THEN
            save(look-aheads);
        look-ahead := Token
    END; (* procedure insert *)

PROCEDURE save(Tokens)
    BEGIN (* save tokens in buffer *)
        Buffers := Tokens
    END; (* procedure save *)
```

The parser-driver is modified as follows:

```
SWITCH (Action) OF
    CASE Shift:
        IF no look-ahead token THEN
            Symbol := Lexicalanalyzer()
        ELSE Symbol := look-ahead;
        State := Nextstate(State, Symbol);
        restore(Symbol);
    END; (* case shift *)

    CASE Reduce: (* reduce by appropriate production *)
    END; (* case reduce *)

    CASE Accept: (* halt, accepting *)
    END; (* case accept *)

    CASE Error: (* halt, rejecting *)
    END; (* case error *)

PROCEDURE restore(Symbol)
    BEGIN (* check if buffer is empty *)
        IF Buffer <> Empty THEN
            BEGIN
                Symbol := Buffer;
                Buffer := Empty
            END
    END; (* procedure restore *)
```