Instruction Selection by Attributed Parsing

Mahadevan Ganapathi
Charles N. Fischer

Technical Report No. 84-256

February 1984

Charles Fischer is affiliated with the Computer Sciences Department, University of Wisconsin, Madison. This research has been supported in part by NSF grant MCS78-02570.

Inst ruction Selection by Attributed Parsing

Mahadevan Ganapathi
Charles N. Fischer

Technical Report No. 256

February 1984

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

Affix grammars are used to describe the instruction-set of a target architecture for purposes of compiler code generation A code generator is obtained automatically for a compiler using attributed parsing techniques. A compiler built on this model can automatically perform most popular machine-dependent optimizations, including peephole optimizations. Implementations of code generators based on this model exist for the VAX [1] -11, iAPX[2] -86, Z-8000[3] , PDP[4] -11 and IBM-370 architectures.

Key Words and Phrases: Compiler-Compiler, Code-Generator-Generator, Code Generation, Machine Description, Intermediate Representation, Machine-dependent Optimization, Code Optimization.

1 VAX is a trademark of Digital Equipment Corporation.

2 iAPX is a trademark of Intel Corporation.

3 Z-8000 is a trademark of Zilog Inc.

4 PDP is a trademark of Digital Equipment Corporation.

1. Table-Driven Code Generation

Given the diversity of architectures commonly available, and the expense inherent in the development of quality software, it is imperative that programs be as transportable as possible. For most programs, this goal is accomplished by using a widely available and well standardized high level language such as Pascal or C. Transporting compilers raises special problems because the machine for which they are to generate code is often mixed *with* the compiler. Considerable time and software investments are involved to produce compiler code generators. The proliferation of programming languages and computer architectures has created the need for automating the code-synthesis phase in compilers [SIGN83a].

To enhance the transportability of compilers, a number of investigations have taken place in the past decade. Previous research in automatic code generation can be broadly classified into three categories: Interpretive code generation, pattern-matched code generation and table-driven code generation. Interpretive code generation approaches generate code for a virtual machine and then expand into *real* target code. Pattern-matched code generation approaches separate the machine description from the code generation algorithm. Table-driven code generation approaches are automated enhancements of pattern-matched schemes. They employ a formal machine description and use a code-generator-generator to produce code generators automatically. For an extensive review and critique of research in automatic code generation, the reader may refer [SURV82,83].

In table-driven code generation schemes, pattern matching is used to replace interpretation with case analysis. Instruction patterns may be tree-structured [CATT80, WULF80] or linear as used by Graham and Glanville [GLAN78] and Ganapathi and Fischer [POPL82]. Correspondingly, pattern matching is performed by heuristic search [FRAS77, CATT80] or parsing [GLAN77, GANA80]. Glanville and Graham made a significant breakthrough; while their approach is not a complete solution, it is a big step towards automating the code generation phase in compilers: The code generation model presented in this paper is an extension of the Graham-Glanville approach. Semantic attributes, disambiguating predicates and a semantic evaluation framework are used to provide formalized attribute processing and automated semantic handling. The target machine is described by Affix grammars [KOST71] instead of context free grammars [AHOU73]. Such descriptions are used to formally specify the semantics of the instruction-set of the target architecture. Intuitively, each symbol in a context free grammar is allowed to have a fixed number **of** associated values, termed **attributes,** whose domains may be finite or infinite. The attributes associated with a given symbol may be **synthetic or inherited.** Synthetic attributes, which are prefixed by a $\uparrow$, are used to pass information up a parse tree. Inherited attributes are prefixed with a $\downarrow$ and are used to pass information down a parse tree. Each context free production has an associated set of attribute evaluation functions. The function symbols can be partitioned into **predicate** symbols and **action symbols.** Predicates control applicability of productions. They use semantic information from attributes to control parsing. Action symbols compute new attribute values. They replace traditional semantic functions invoked upon each reduction during parsing. 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.

Affix grammar driven code generation clarifies many of the decisions in code synthesis: storage allocation, instruction selection, machine-dependent and peephole optimization. Code generation is performed by attributed parsing. Semantics and context are used to do all attribute evaluation in a single bottom-up pass. When transforming the intermediate representation of a program, each transformation causes code generation action. Transformations usually require satisfying conditions on semantic attributes of symbols (e.g., occurrences of registers, constants in a certain range). The results of transformations become non-terminals and their attributes in the grammar.

In this endeavor, the particular goals are:

- Design affix grammars for target architectures and generate tables automatically from these descriptions. Real computers have numerous instructions that can be used to effect the same result (e.g., add, add immediate, increment by a small constant). Predicates are used to define how and when a given instruction would be selected. By controlling production-application, predicates are used to block productions when restricted by the target architecture.

- Evaluate the human effort involved in the construction of productions and attribute predicates in light of architectural restrictions on the programming model. Of particular interest are the impact of complicated and non-orthogonal addressing architectures, irregular, inflexible and overlapping register architectures and segmentation models on code generation

- Use special addressing modes and instructions of target architectures to produce high quality target code. Predicates are used as a tool to incrementally add machine-dependent and peephole optimizations [MCKE70, WULF75] and thus improve target code quality. These optimizations are incorporated in a cohesive manner as part of a well understood parsing process.

2. Attributed Prefix Intermediate Representation

In a compiler written for a single programming language to run on a single target machine, an intermediate form of program code is used primarily to allow optimization. Examples of popular intermediate forms are quadruples, triples, tree representation of programs and direct acyclic graphs [AHOU77].[6] Traditionally, each has been used during different phases of compilation. They have served as a suitable model for numerous tasks including global flow analysis, loop optimizations, expression optimizations, global register allocation, frequency reduction and code generation. Unfortunately, these conventional intermediate forms are inadequate to serve the needs of compiler retargetability. Quadruples and triples have been conventionally useful for program optimization. They require explicit temporary specification. The number of temporaries required for evaluation is

[6]SIGN83b provides a bibliography on intermediate representations.

often a machine-dependent issue. Furthermore, N-ary operations such as array indexing and procedure calls must be realized as multiple unary and binary operations. N-tuples alleviate this problem but include the drawbacks of explicit temporary specification. Trees allow a more complete representation of programming language semantics but do not promote efficient pattern matching during code generation. They often contain redundant information in parse-tree form. Graphs are not a good model for code generation. Cycles must be eliminated and pattern matching poses a greater problem. Prefix and postfix notations by themselves are inadequate for complete representation of programming language semantics. However, the incorporation of attributes in the intermediate representation provides a fairly adequate representation of semantics. The rationale behind this choice, its relative standing with respect to other proposals in the literature and the detailed design of this attributed prefix intermediate representation (IR) is explained in [SPE84]. The main goal in this design is to minimize the effort required to retarget compiler code generators.[7] The design of an intermediate representation is critical to compiler portability and code generation efficiency. This efficiency issue includes both the efficiency of the code generation algorithm and that of the object code produced by the code generation algorithm. The use of a prefix representation facilitates efficient pattern matching between IR and the description of the target machine instruction-set. String matching algorithms can be used to perform pattern matching. Typical parsers such as YACC [JOHN75] use very simple drivers and are efficient (i.e., linear time algorithms) and well understood. Pattern matching in a tree is not as well understood as parsing. Most tree-matching algorithms rely on heuristics and are not provably correct. Our richer view of an intermediate representation makes IR specifically susceptible to various kinds of on-the-fly processing using a single pass pattern matching scheme.

Attributes are useful in guiding the code generator to emit efficient target code. They carry forward semantic information such as:

- environment options -- choice of display/access mechanism, choice of offsets whether positive, negative or both, direction of run-time stack growth, first argument of procedure parameters pushed last or first.

- storage binding -- type, size and scope of a variable.

- code optimization -- execution speed and object code space, context, state information of basic blocks.

At code generation time, these attributes control pattern matching of IR with the machine description. The attribute domains of operators and operands are determined by the structure of the high level language. Consider the following fragment of Pascal code:

```
PROGRAM Example (input, output);
CONST
```

[7]STAN84 describes a recent effort for DEL architectures.

```
                numitems = IO;
        VAR
                index : INTEGER;
        BEGIN
                index := 0;
                REPEAT
                        index := index + 1;
                UNTIL   index > numitems;
        END.
```

The IR translation of this program reads:

```
        : Example ↑2
                : input ↑Parameter tpointer ↑1
                : output ↑Parameter tpointer ↑1
                : index ↑Global tlonginteger ↑1
                := index0
                repeatlabel
                        := index + index 1
                        < = index 10 repeatlabel
```

IR variables are assigned storage before code selection. The storage binding phase expands this IR into primitives that transform the string representation of a variable into access mechanisms involving the display. In the above example, the global variable index and the constants 0, 1 and 10 are transformed as follows:

```
: = Datumti Datum↑0 Label↑L30  : = Datumti + Datumti Datumtl
< = Datumti Datum↑10 Label↑L30
```

The attribute variable *i* includes address information and the datum size for index; long on the VAX and *word* on the iAPX-86 and the PDP-11. The mapping of this intermediate form to target code is described in the section on Instruction Selection.

3. Machine Description

Retargetable code generation is achieved by separating the code generation algorithm from the target machine data that drive it. The correspondence between semantic primitives in the IR and target machine operations is built by enforcing conventions in the target machine specification. For purposes of pattern matching and instruction selection, the instruction set of the target architecture is represented as a set of affix-grammar productions. These productions form the input to a program that generates a code generator for the target machine. 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 appearing with synthetic attributes only. The RHS contains:

- terminals with synthetic attributes (↑),
- non-terminals with synthetic attributes (↑),
- disambiguating predicates (italicized) with inherited attributes (↓) and boolean operations

on them, and

- 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. Reference to a piece *elem* of an attribute variable var is done by using the '.' operator *as* ***var.elem***. The same attribute variable may appear on both LHS and RHS of a production. Since the LHS has synthetic attributes, the RHS must synthesize these attributes and the LHS' synthetic attributes must appear as synthetic attributes in RHS symbols: terminals, non-terminals and action symbols. 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↑a → Addressfa,

the attribute variable *a* is copied from symbol Address to symbol 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. For example, the production

Byte↑a → Addressfa *IsByte↓a*

is used only if Address has attributes that show it is a byte. A production is applicable only if the boolean function of its component disambiguating predicates evaluates 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 but not both with selection at code generation time. Our experience suggests that a single linear ordering usually suffices e.g., space $\underline{C}$ time. However, in general, different implementations of the same instruction-set have different speeds for instructions, operands and cache effects. In such cases, a non-linear ordering of predicates will be necessary. The kinds of productions needed for an entire code generator can be broadly classified into addressing-mode productions, instruction-selection productions and transfer productions.

Addressing-mode productions

Each addressing-mode production has an RHS specifying an addressing mode of the target architecture in the language of IR. The production creates the proper machine address, in an action symbol. For example, the following production is used to specify a displacement addressing mode on the VAX-1 1:

Address↑a → $ Datumfb Datum↑c *displace@* A *register↓c* ADDR↓b↓c↑a

$ is a grammar terminal that denotes the indexing operator. Datum and Address are grammar non-terminals with attributes specifying the value of the displacement *b*, base register c and the synthesized addressing mode a respectively. The attribute variable *b* specifies the machine data type and offset from a frame pointer. The attribute variable c specifies the index (or display) register of the IR variable. These attributes are determined when IR variables are bound to locations in the target machine. The predicates *displace* and *register* specify target architecture restrictions on using this addressing mode. Examples of such restrictions are IBM-370 restrictions on displacement and base register use, iAPX-86 restrictions due to segmentation and index register use, and in general if c happens to be a memory location instead of a register on most machines. If the predicate function evaluates to *false,* recognition of this addressing mode production is **blocked.** Consequently, a subsequent production is matched that forces, via action symbols, to satisfy target architecture restrictions on the use of this addressing mode.

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:

storage class: memory, frame, stack, base register,
an offset from the base register,
levels of indirection,
an index register (if any),
the datum size and
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-l 1, iAPX-86, Z-8000, IBM-370 and the PDP-1 1, 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, since the index register and the base register cannot be used simultaneously, the index-register field will never be used. Addressing-mode productions reflect addressing modes supported by the target machine. If the target machine does not support particular 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.

The section on machine-dependent and peephole optimization discusses productions to subsume arithmetic within the addressing modes of the target architecture. Addressing mode side-effects such as predecrement and postincrement are also discussed in the same section.

Inst ruction-selection productions

Each production has an RHS specifying a pattern in the IR and the corresponding code sequence to be emitted on a match. The LHS may be an explicit result location, such as, 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. In the case of multiple results (e.g., quotient and remainder, arithmetic result and condition-code side effects), the LHS specifies the data type of the primary result. Secondary results are synthetic attributes to the LHS. Consider addition on the VAX-I 1/780. There are two-address and three-address add op-codes. Furthermore, the increment (or decrement) instruction can be used for adding one (or subtracting one). For a byte datum, these forms of addition are expressed as follows:

```
Byte↑r↑cc → + Byte↑1 ↑cc1 Byte↑r↑ccr ¬ Busy↓r
                                       EMIT↓'incb'↓r↓0↓0↑cc
Byte↑r↑cc → + Byte↑r↑ccr Byte↑1 ↑cc1 ¬ Busy↓r
                                       EMIT↓'incb'↓r↓0↓0↑cc
Byte↑r↑cc → + Byte↑-1 ↑cc1 Byte↑r↑ccr ¬ Busy↓r
                                       EMIT↓'decb'↓r↓0↓0↑cc
Byte↑r↑cc → + Byte↑r↑ccr Bytef -1 ↑cc1 ¬ Busy↓r
                                       EMIT↓'decb'↓r↓0↓0↑cc
Bytefrfcc → + Byte↑a↑cca Byte↑r↑ccr ¬ Busy↓r
                                       EMIT↓'addb2'↓a↓r↓0↑cc
Byte↑r↑cc → + Bytefrfccr Byte↑a↑cca ¬ Busy↓r
                                       EMIT↓'addb2'↓a↓r↓0↑cc
Byte↑r↑cc → + Byte↑a↑cca Byte↑b↑ccb
                                       TEMP↓'byte'↑r
                                       EMIT↓'addb3'↓a↓b↓r↑cc
```

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), its attributes (r, cc) specify the data type, address of the result and condition-code setting of the selected instruction respectively. Similarly, the third and fourth productions specify the addition of *-1* to *r.* The fifth and sixth 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 TEMP. This action symbol returns a *temporary location,* such as a free register, for purposes of expression evaluation. An addition of two IR data in byte format will match the RHS of one of these productions. The last production is a **default** production and must be supplied. If this production is omitted, the code generator may block when attempting to produce target code. If the other productions are omitted, the code generator will produce inefficient code in certain cases but it will not block. The choice of the RHS is determined by attribute values and the disambiguating predicates. If an operand is 1 (or -1) then an *incb* (or *decb*) instruction is selected. Productions five through seven handle addition of a constant other than 1 (and -1) as well as variables. The predicate Busy evaluates to *false* if the value of its attribute variable need not be preserved after the operation. Without inherited attributes to non-terminals, this information (i.e., the desired left context) must be available at a constant offset from the top of the parser stack (c.f., Section 5). Consequently, a two-address *addb2* is selected. If *Busy* evaluates to *true,* then a three-address *addb3* is selected. Other productions are used to perform arithmetic in addressing modes of the target architecture, such as *movab* with indexing on the VAX. This technique is illustrated in a subsequent section on Optimization.

Transfer Productions

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:

- Destructive operations: Many machine operations, such as two-address instructions, destroy the contents of a participating operand. For example, on the PDP-11, add *A, B* destroys the contents of location *B*. Thus, to implement C *:* = *A* + *B*, *B* or *A* must be in a temporary location before addition.

- Data-type conversion: Most machine op-codes operate only on operands of identical data types except where data types are explicit in the instruction or in the data. 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 conversions, some machines provide a special conversion instruction, such as, *cvtbw* on the VAX-l 1 whereas other machines might require a sequence of instructions. This sequence of instructions is specified by the instruction-set describer.

- 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-8000 and iAPX-86, no memory-to-memory arithmetic is possible. On the PDP-1 1 and IBM-370 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-1 1, 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-l 1, the following transfer production is used:

```
Long↑r↑cc → Wordfafcca CheckConvert↓a↓'word'↓'long'
                                TEMP↓'long'↑r
                                EMIT↓'cvtwl'↓a↓r↓0↑cc
```

The predicate *CheckConvert* is used to check if conversion from *word* to *long* format is really needed in the current context of operation. Once again, context is determined by interrogating the parser stack. *CheckConvert* will evaluate to true if the variable represented by the attribute *a* must be converted to a *long* datum on the target machine. This conversion usually takes place when mixed-mode arithmetic is performed on non-tagged architectures. The absence of *transfer productions* **may** force code generators to block while processing a syntactically correct IR input. The inclusion **of** *transfer productions* may cause loops in the code-generator automaton when data type conversions are performed more than once for the same variable. For example, a *word* variable may be converted to a *long* variable and then subsequently converted back to a *word* variable without consuming any IR input. The predicate *CheckConvert* is therefore used to check if conversions are performed more than once for the same variable; thus, avoiding a potential looping configuration of the code generator.

4. Inst ruction Selection by Attributed Parsing

The affix 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 affix-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. The CGG constructs a **context-**sensitive parser. The parser constructor is a generalization of context-free parsing methods **that** accepts a useful class of affix 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 deterministic context-free parsers. Both top-dew-n and bottom-up parsers have proved attractive to language implementors since they organize the translation phase of compilers. This section discusses the use of attributed 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). 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 can be associated directly with the production whose prediction it will determine. Since, before prediction, very little information is available on the operands, many (unbounded) lookaheads are required to select the proper template. Predicates need to consider the non-terminal on top of the parser stack along with these look-aheads. In contrast, bottom-up parsing is far 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 by resolving *shift-reduce* and *reduce-reduce* conflicts. However, affix grammars must be restricted in the following ways to make them suitable for one pass left-to-right attributed bottom-up processing.

- 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 [WATT77, FISC83]. For LR(k) or SLR(k) grammars, to determine positions suitable for calling semantic action routines, the reader is referred to [PURD80].

- 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 constructed. 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.

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. The system associates predicates with states when it builds the parse table. In practice, the disambiguating predicates of each production are usually written as the productions are designed, i.e., the predicates are production, and not state, oriented. 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 by resolving shift-reduce and reduce-reduce conflicts. 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. 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. Attributed parsing for our implementation (CG) is limited to single-pass left-to-right evaluation of attributes. The LHS of a production does not have any inherited attribute. Thus, information is never passed down the tree or to left siblings. The attributed bottom-up parser with disambiguating predicates employs the standard LR(k) parsing loop with added code to manipulate attributes. Using two stacks, the control stack and the attribute stack, aids conceptual clarity. The prototype code generator uses about ten attributes covering many architectures. Since the set of attributes is relatively small, 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$\uparrow$2 will only match a datum with an attribute value of 2, whereas **Datum↑a** can match a datum with any attribute value. 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.

5. Examples of Inst ruction Selection

In this section, we illustrate the translation of the IR example in Section 2 to target code for the iAPX-86, PDP-11 and VAX-l 1 using attributed parsing. The following lines trace the parsing process. They describe every step of production recognition, showing the input to the code generator and the transformed output for the iAPX-86. The production that is matched at each step is explained and the code that is generated upon production recognition is enclosed within {}. The symbol ⊢ denotes the point of a *shift* action in the IR string. The symbol ⊣ denotes the point of a *reduce* action in the IR string. On a reduction, predicate symbols for the configuration set are invoked in order till a match is found.

[l] ⊢ : = Datumfi DatumfO **Label↑L30** : = Datumfi + Datumfi Datumfl
< = DatumTi **Datum↑10 Label↑L30**

[2] : = ⊢ Datumfi **Datum↑0 Label↑L30** : = Datumfi + DatumTi DatumTl
< = Datumfi **Datum↑10** LabelfL30

[3] : = DatumTi ⊣ DatumfO LabelfL30 : = Datumfi + Datumfi Datumfl
< = DatumTi **Datum↑10 Label↑L30**

The following addressing mode production is selected.

Address↑ index → DatumTi **ADDR↓i↓0↑index**

[4] : = Addressfindex ⊣ DatumfO **Label↑L30** : = Datumfi + Datumfi DatumTl
< = Datumfi **Datum↑10 Label↑L30**

The first predicate for this configuration set *IsByte↓index* evaluates to *false.* The next predicate *IsWord↓index* evaluates to *true.*

[5] : = **Address↑index** *IsWord↓index* ⊣ **Datum↑0 Label↑L30** : = Datumfi + Datumfi DatumTl
< = DatumTi **Datum↑10 Label↑L30**

The following production is selected.

Word↑x → Address↑x *IsWord↓x*

[6] : = Wordfindex ⊢ DatumfO **Label↑L30** : = DatumTi + DatumTi DatumTl
< = DatumTi **Datum↑10 Label↑L30**

[7] : = Word↑index Datum↑0 ⊣ Label↑L30 : = DatumTi + DatumTi Datum↑1
< = DatumTi Datum↑10 LabelfL30

[8] : = Wordfindex Datum↑0 *CheckConvert↓0↓ 'constant'↓'word'* ⊣ Label↑L30
: = DatumTi + Datumfi Datum↑1 < = Datumfi Datum↑10 Label↑L30

The following transfer production is matched.

Word↑c → Datumfc *CheckConvert↓c↓'constant'↓* 'word'

[9] : = Word↑index Word↑0 ⊣ LabelfL30 : = Datumfi + Datumfi Datum↑1
< = DatumTi Datum↑ 10 Label↑L30

[10] : = Word↑index Word↑0 ¬ *Same↓index↓0* A ¬ *Memory↓index↓0* ⊣ LabelfL30
: = DatumTi + Datumfi Datumfl < = Datum↑i DatumTlO Label↑L30

The following iAPX-86 production is matched.
Same evaluates to true if its attributes are equivalent locations.
Memory evaluates to true if its attributes are both memory locations.
CCMOV is the attribute specifying the condition codes set by the iAPX-86 MOV instruction.
It is synthesized by the action symbol EMIT.

Instruction↑ccmov → : = Wordfx Word↑y ¬ *Same↓x↓y* A ¬ *Memory↓x↓y*
EMIT↓'mov'↓x↓y↓0↑ccmov

[11] Instructionfccmov {mov index, 0} ⊢ LabelfL30 : = Datumfi + DatumTi Datumfl
< = Datumfi Datumf 10 Label↑L30

[12] Instruction↑ccmov {mov index, 0} LabelfL30 ⊣ : = DatumTi + Datumfi Datum↑1
< = DatumTi Datum↑1 0 Label↑L30

The following production is matched.

Instruction↑ccnotset → Label↑n EMIT↓'label'↓n↓0↓0↑ccnotset

[13] Instructionfccmov {mov index, 0} Instruction↑ccnotset {L30:} ⊢ : = Datum↑i
+ Datumfi Datumfl < = Datumfi DatumTlO Label↑L30

[14] Instructionfccmov {mov index, 0} Instruction↑ccnotset {L30:} : = ⊢ Datumfi
+ Datumfi Datumfl < = Datumfi DatumTlO LabelfL30

[15] Instructionfccmov {mov index, 0} Instructionfccnotset {L30:} : = Datumfi ⊣
+ DatumTi Datumfl < = Datumfi DatumTlO Label↑L30

The following addressing mode production is selected.

Addressfindex → Datumfi ADDR↓i↓0↑index

[16] Instructionfccmov {mov index, 0} InstructionTccnotset {L30:} : = Addressfindex ⊣

+ DatumTi Datumfl <= Datumfi DatumTlO LabelfL30

[17] **Instruction↑ccmov {mov index, 0}** Instructionfccnotset **{L30:} : =** Addressfindex
Is Word↓index ⊣ + DatumTi Datum 1 < = DatumTi Datum↑10 **Label↑L30**

The following production is selected.

Word↑x → Addressfx *IsWord↓x*

[18] Instructionfccmov **{mov index, 0}** Instructionfccnotset **{L30:}** : = Wordfindex ⊢
+ Datumf i Datum↑1 < = DatumTi DatumTlO **Label↑L30**

[19] **Instruction↑ccmov {mov index, 0}** Instructionfccnotset **{L30:}** : = **Word↑index**
+ ⊢ DatumTi Datum↑1 < = DatumTi DatumTlO LabelfL30

[20] Instructionfccmov **{mov index, 0}** Instructionfccnotset **{L30:}** : = **Word↑index**
+ Datumfi ⊣ Datumfl < = DatumTi **Datum↑10 Label↑L30**

The following addressing mode production is selected.

Addressfindex → DatumTi **ADDR↓i↓0↑index**

[21] Instructionfccmov **{mov index, 0} Instruction↑ccnotset {L30:} : =** Wordfindex
+ Addressfindex ⊣ **Datum↑1** < = DatumTi DatumTlO LabelfL30

[22] Instructionfccmov {mov **index, 0}** Instructionfccnotset **{L30:}** : = Wordfindex
+ Addressfindex *IsWord↓index* ⊣ **Datum↑1** < = DatumTi **Datum↑10** LabelfL30

The following production is selected.

Wordfx → Addressfx *IsWord↓x*

[23] ·Instructionfccmov **{mov index,** 0} **Instruction↑ccnotset { L30:} : =** Wordfindex
+ Wordfindex ⊢ Datum↑1 <= DatumTi DatumTlO **Label↑L30**

[24] Instructionfccmov **{mov index, 0}** Instructionfccnotset **{L30:} : = Word↑index**
+ Wordfindex Datumf 1 ⊣ <= Datumfi **Datum↑10 Label↑L30**

[25] Instructionfccmov **(mov index, 0} Instruction↑ccnotset {L30:}** : = **Word↑index**
+ Wordfindex Datumfl *CheckConvert↓1↓'constant'↓'word'* ⊣
< = Datumfi **Datum↑10 Label↑L30**

The following transfer production is matched.

Word↑c → Datum↑c *CheckConvert↓c↓'constant'↓'word'*

[26] Instructionfccmov **{mov index, 0} Instruction↑ccnotset {L30:}** : = **Word↑index**
+ Wordfindex **Word↑1** ⊣ < = DatumTi **Datum↑10 Label↑L30**

[27] Instructionfccmov **{mov index, O}** InstructionTccnotset **{L3O:}** : = Wordfindex
+ Wordfindex Word↑1 ¬ *Busy↓index* ⊣ < = Datumfi Datum↑10 Label↑L30

The following production is matched.

Word↑index↑ccinc → + Wordfindex Word↑1 ¬ *Busy↓index*
EMIT↓'inc'↓index↓0↓0↑ccinc

This production describes a special purpose *increment* instruction, applicable only if the first operand's contents need not be saved and the second operand is the constant 1. The predicate Busy evaluates to false if the contents of its attribute location need not be saved. To determine this condition, the desired left context is known by interrogating the first and second symbol below the + operator on the attributed parser stack. In this case, these symbols are Word↑index and : = respectively. Thus, the variable *index* is the target of the assignment. Consequently, *Busy↓index* evaluates to *false* and the *inc* production is selected. The action symbol EMIT synthesizes the condition codes, *ccinc,* set by the iAPX-86 *inc* instruction.

[28] Instruction↑ccmov **{mov index, O}** Instruction↑ccnotset **{L3O:}** : = Word↑index
Wordfindexfccinc **{inc index}** ⊣ < = Datum↑i Datum↑10 LabelfL30

The following production is recognized.

Instruction↑cc → : = Word↑x↑ccx Wordfxfcc

This production is similar to that in step [IO] except that no code is emitted. ¬ *Same↓index↓index* evaluates to *false*. Consequently, this production is matched and no iAPX-86 target code is emitted.

[29] Instruction↑ccmov **(mov index,** 0} Instruction↑ccnotset **{L3O:}** Instructionfccinc
(inc index} ⊢ < = Datumfi Datum↑10 Label↑L30

[30] Instructionfccmov **{mov index, O}** Instructionfccnotset **{L3O:}** Instruction↑ccinc
(inc index) < = ⊢ Datum↑i Datum↑10 Label↑L30

[31] Instructionfccmov **{mov index, O}** InstructionTccnotset **{L3O:}** Instructionfccinc
{inc index) < = Datumfi ⊣ Datum↑10 Label↑L30

The following addressing mode production is selected.

Addressfindex → Datumfi ADDR↓i↓0↑index

[32] Instruction↑ccmov **{mov index, O}** Instructionfccnotset **{L3O:}** Instructionfccinc
{inc index} < = Addressfindex ⊣ Datum↑10 Label↑L30

[33] Instructionfccmov **{mov index,** 0} Instructionfccnotset **{L3O:}** Instructionfccinc
(inc index) < = Address↑index *IsWord↓index* ⊣ Datum↑ 10 Label↑L30

The following production is selected.

Wordfx → Address↑x *IsWord↓x*

[34] Instructionfccmov **{mov index, 0}** Instruction↑ccnotset {L30:} Instructionfccinc **{inc index}** < = Word↑index ⊢ Datum↑10 Label↑L30

[35] Instructionfccmov **{mov index,** 0} Instructionfccnotset **{L30:}** Instructionfccinc **{inc index} < =** Word↑index Datum↑10 ⊣ LabelfL30

The following transfer production is matched.

Wordfc → Datum↑c *CheckConvert↓c↓'constant'↓'word'*

[36] Instructionfccmov **{mov index, 0}** Instruction↑ccnotset {L30:} Instructionfccinc **{inc index}** < = Word↑index Word↑10 ⊣ LabelfL30

The following instruction-selection production is matched.

Branchf'jle'fcccmp → < = Word↑a Wordfb EMIT↓'cmp'↓a↓b↓0↑cccmp

[37] Instruction↑ccmov **{mov index, 0}** InstructionTccnotset {L30:} Instruction↑ccinc **{inc index}** Branch↑'jle'↑cccmp **{cmp index,** 10) ⊢ Label↑L30

[38] Instruction↑ccmov **{mov index,** 0} Instruction↑ccnotset **{L30:}** Instructionfccinc **{inc index}** Branchf'jle'fcccmp **{cmp index, 10}** Label↑L30 ⊣

The following production is selected.

Instruction↑ccbr → Branch↑brcode↑cc Labelflab EMIT↓brcode↓lab↓0↓0↑ccbr

[39] Instruction↑ccmov **{mov index, 0}** Instructionfccnotset {L30:} Instruction↑ccinc **{inc index}** Instruction↑ccbr **{cmp index, 10) {jle L30}**

The goal symbol of the machine grammar is Codegenerator and it is described as:

Codegenerator → Instructions

Instructions → Instructions Instruction
Instructions → Instruction

The iAPX-86 target code for the IR example in Section 2 reads:

```
move index, 0
L30:
inc index
cmp index, 10
jle L30
```

The PDP-11 target code for the same IR example is slightly different. The *clr* instruction is selected instead of a mov. Reconsider step [10] above. The PDP-11 grammar contains the following production:

Instruction↑ccclr → : = Wordfx Word↑0 ¬ *Same↓x↓0* EMIT↓'clr'↓x↓0↓0↑ccclr

Consequently, step [1 1] becomes:

[11] Instruction↑ccclr {clr index} ⊢ LabelfL30 : = Datum↑i + Datum↑i Datumfl
< = Datum↑i Datum↑10 Label↑L30

Similarly, the short-branch instruction *ble* is selected using the following PDP-11 productions:

Branch↑'jle'↑'ble'↑cccmp → < = Word↑a Wordfb EMIT↓'cmp'↓a↓b↓0↑cccmp

Instructionfccbr → Branch↑longbr↑shorbr↑cc Label↑lab *Shortdistance↓lab*
EMIT↓shortbr↓lab↓0↓0↑ccbr

Thus, the PDP-11 target code for the IR example reads:

```
clr index
L30:
inc index
cmp index, $12      /octal 12 for 10
ble L30
```

The VAX-1 1 machine grammar has a completely different production to describe an *add-one-and-branch-less-than-or-equal-to* instruction. The datum *index* is mapped to a *long* data type on the VAX-1 1 . The steps [26] till the end are replaced by a single match of the following production:

Instruction↑ccaob → : = Long↑a + Longfa Long↑1 < = Long↑a Long↑b Labelflab
Shortdistancellab
EMIT↓'aobleq'↓b↓a↓lab↑ccaob

The predicate *Shortdistance* evaluates to *true* if its attribute is a short-distance target label for an immediate branch. In this example *Shortdistance↓L30* evaluates to *true.* Thus, the *aobleq* instruction is selected and the VAX-l 1 target code is substantially different from those of the iAPX-86 and the . PDP-11:

```
clr index
L30:
aobleq $10, index, L30
```

6. Integrated Peephole Optimization

Attributes are used in more complex ways to make optimization decisions Incorporation of these optimizations is advantageous for code efficiency. Constant folding, strength reduction, auto-decrement/increment and subsumption of address arithmetic within addressing modes are some of the optimizations that are incorporated as grammar productions Others such as delaying stores into memory, register tracking to avoid redundant loads and stores, and code hoisting are incorporated as a *coroutine* to the instruction selection process within a *basic block* [AHOU77]. The use of an *integrated* peephole optimizer as part of code selection, reduces the need for separate optimization phases. Thus, need for a separate peephole optimizer is reduced if not eliminated. The rationale behind an integrated peephole optimizer instead of a separate package, such as an algebraic simplifier on Register-Transfer-Level descriptions [DAVI80, GIEG83], is explained below.

- In a separate peephole optimizer, type bindings are lost. It becomes harder to identify pointers and negative numbers. Thus, strength reduction of multiplication by shifting is difficult. For example, the VAX supports an *arithmetic shift* operation on *long* data types only.

- Since IR is free of machine-specific idiosyncracies, certain peephole optimizations can be done at the IR level without requiring live/dead context information [TANN82]. For example, on the PDP-11, *add # 1, r0* sets the *carry* condition-code bit whereas *inc r0* does not set the *carry* bit. In order to replace this *add* with an *inc,* a separate peephole optimizer package must analyze each occurrence of *add # 1, r0* to determine if the *carry* bit is used later. If the *carry* bit is used later, then *inc* cannot be selected.

- Auto-increment/decrement optimization can be performed within a *logically* adjacent window as opposed to a *physically* adjacent window.

- In spite of global register allocation strategies [CHAI82, LEVE81], local register management is essential for tracking register aliasing, register pairing and overlapped sharing (e.g., AL, AX on iAPX-86; RLO, RO, RRO, RQO on Z-8000). A separate peephole optimizer package can conflict with the local register manager [RUDM79, CROW82].

- Sometimes, strength reduction can affect register management. For example, on the iAPX-86, the AX accumulator is *sacred* for many instructions and in particular for multiplication. Consequently, before multiplication, the previous contents of AX and its corresponding register pair DX must be saved. However, if an *add* is selected instead of a *multiply by 2* or if a *shift* is selected instead of a *multiply by powers of* 2, then the AX and DX registers need not be saved. This optimization, when done in a separate peephole optimizer package, would be late for improving register management.

Constant optimizations and constant folding

It is possible to optimize addition and subtraction of zero, multiplication and division by one and constant folding by expressing them as affix grammar productions. The following iAPX-86 productions describe some of these cases as examples

Byte↑r↑cc → · Byte↑r↑ccr Byte↑0↑cc

Byte↑r↑cc → / Byte↑r↑ccr Bytefl ↑cc

Byte↑r↑cc → / Byte↑a↑cca Byte↑b↑cc *Constant↓a A Constant↓b*
FOLD↓'/'↓a↓b↑r

Strength reduction

The VAX-l 1 has a three-address arithmetic shift *ashl* and both two and three-address multiply instructions *mull2, mull3* that can be exploited in its affix grammar:

```
Long↑r↑ccash → * Long↑a↑cca Long↑r↑ccr
                                   ¬ Busy↓r A Power2↓a A Positive↓r
                                   EMIT↓'ashl'↓$log_2$a↓r↓r↑ccash

Longfrfccash → * Long↑r↑ccr Long↑a↑cca
                                   ¬ Busy↓r A Power2↓a A Positive↓r
                                   EMIT↓'ashl'↓$log_2$a↓r↓r↑ccash

Long↑r↑ccmul2 → * Long↑a↑cca Long↑r↑ccr
                                   ¬ Busy↓r
                                   EMIT↓'mull2'↓a↓r↓0↑ccmul2

Longfrfccmul2 → * Long↑r↑ccr Long↑a↑cca
                                   ¬ Busy↓r
                                   EMIT↓'mull2'↓a↓r↓0↑ccmul2

Longfrfccash → * Long↑a↑cca Long↑b↑ccb
                                   Power2↓a A Positive↓b
                                   TEMP↓'long'↑r
                                   EMIT↓'ashl'↓$log_2$a↓b↓r↑ccash

Long↑r↑ccash → * Long↑b↑ccb Long↑a↑cca
                                   Power2↓a A Positive↓b
                                   TEMP↓'long'↑r
                                   EMIT↓'ashl'↓$log_2$a↓b↓r↑ccash

Long↑r↑ccmul3 → * Long↑a↑cca Long↑b↑ccb
                                   TEMP↓'long'↑r
                                   EMIT↓'mull3'↓a↓b↓r↑ccmul3
```

Attributes and predicates are very useful in selecting between optimization for space and optimization for time when they conflict. The applicability of the arithmetic shift production can depend further on whether optimization for time is to be preferred over optimization for space. The two-address multiply on the VAX-l 1 occupies less space than an arithmetic shift which needs three addresses. But an arithmetic shift is faster than a multiply instruction. The user may set options such as *optimization for space preferred to optimization for time* as an attribute *space* to the multiply operator *. The grammar writer must define a predicate such *as Timepreference* that evaluates to *false* if the attribute of * is *space.* This predicate is specified in conjunction with other predicates for all *ashl* productions that appear before the *mull2* productions

Similarly, if optimization for time is preferred over optimization for space, multiplication by 3 may be strength reduced to an *arithmetic shift* followed by an *add.*

Fancy instructions -

Many target architectures provide special-purpose instructions to yield optimized target code. Examples of such instructions are: *subtract-one-and-branch* on the PDP-11, *subtract-one-and-branch-less-than-or-equal* and *add-one-and-branch-less-than-or-equal* on the VAX-l 1. It is not essential to use such instructions in the translation of user programs for the target architecture. However, by using such instructions, compilers can produce efficient representations of user programs. It is possible to represent these fancy instructions as grammar productions with a long right hand side. Extra *blocking predicates* are used to specify the applicability of such productions. For example, consider the *subtract-one-and-branch* series of instructions on the VAX-l 1:

```
Long↑a↑ccsob → : = Long↑a↑cc1 - Long↑a↑cc2 Longfl ↑cc3 > Long↑a↑cc4 Long↑0↑cc5 Labelfn
                                  Shortdistanceln
                                  EMIT↓'sobgtr'↓a↓n↓0↑ccsob

Longfafccsob → : = Long↑a↑cc1 - Long↑a↑cc2 Long↑1 fcc3 > = Long↑a↑cc4 Long↑0↑cc5 Labelfn
                                  Shortdistanceln
                                  EMIT↓'sobgeq'↓a↓n↓0↑ccsob
```

There are no productions to model any *less-than* comparisons in a similar fashion because the VAX-l 1 does not provide corresponding instructions for *subtract-one-and-branch.* Such *optimization productions* are added incrementally to the machine grammar to improve target code quality and to provide *fine tuning* of the object code. They are specified before general *instruction selection* productions so that their predicates are evaluated first. The thrust of this addition is to facilitate incremental development of an optimizing code generator.

Subsuming code within addressing modes

Certain instructions may be subsumed in the addressing modes of the target architecture. Thus, the relevant arithmetic is performed implicitly by the machine hardware. The following examples depict three such cases for the VAX architecture.

- Consider addition within the context of an address calculation. The IR representation could be: = *target Indirect + constant prefixexpression.* If the prefix expression is computed in a register then the addition could possibly be subsumed in the *indexing* addressing mode:

  ```
  Addressfa → Indirect + Long↑d↑ccd Long↑i↑cci Displacement↓d A IndexRegister↓i
                                  ADDR↓d↓i↑a
  ```

  The predicates, *Displacement* and *IndexRegister*, ensure correct applicability of the subsumption production. If the predicate restrictions are not satisfied then the IR is parsed via the usual instruction selection productions and code would be generated *(e.g.,* an *add* in this case). ADDR composes an address attribute *a* with displacement *d* and base register *i.*

- Some machines provide op-codes to obtain the address of a variable. Examples of these instructions are *movab, pushab, movaw, pushaw, moval, pushal* etc. on the VAX and *lea* on the iAPX-86. Using these instructions, certain arithmetic may be subsumed by indexing:

```
Address↑a↑ccpushal → + Long↑d↑ccd Longfifcci Displacementld A IndexRegister↓i
                                  ADDR↓d↓i↑b
                                  TEMP↓'long'↑a Stacktop↓a
                                  EMIT↓'pushal'↓b↓0↓0↑cpushal

Addressfafccmoval → + Longfdfccd Long↑i↑cci Displacement↓d A IndexRegister↓i
                                  ADDR↓d↓i↑b
                                  TEMP↓'long'↑a
                                  EMIT↓'moval'↓b↓a↓0↑cmoval
```

- Many machines, including the VAX and the PDP, contain specialized hardware addressing modes that could potentially yield efficient target 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 by auto-decrement or an add operation can be subsumed in an earlier instruction by auto-increment. Such features can be exploited in affix grammar driven code generation. Consider the following production:

```
Long↑r↑cc → - Long↑r↑ccr Long↑4↑cc Register↓r A ¬ Busy↓r
                                  DELAY↓'subl2'↓4↓r↓0↑cc
```

  This production will be matched only when we find a subtraction of the constant 4 from a *long* format datum in a temporary register. The subtraction from *r* is delayed in hopes of realizing it as a future auto-decrement If the next use of the register *r* is an indirect reference through *r* and the operation uses *long* data, then the auto-decrement addressing mode for r is issued for the current instruction. DELAY is a version of EMIT that delays generation of code till the updated value of its operands is actually necessary in a computation. In this example, if the auto-decrement optimization cannot be done subsequently or if the updated value of the expression is needed, the subtraction is generated. Similarly, if a register is incremented by a constant that is usable in an auto-increment, the previous use of the register-indirect addressing-mode is altered to the auto-increment mode using a *code buffering and register tracking* mechanism outlined in subsequent paragraphs. In the following production, HOIST is another variant of EMIT that attempts to use auto-increment in an earlier instruction:

```
Long↑r↑cc → + Long↑r↑ccr Long↑4↑cc Register↓r A ¬ Busy↓r
                                  HOIST↓'addl2'↓4↓r↓0↑cc
```

Suppressing redundant code

Redundant assignments and condition code settings can be suppressed by grammar productions Examples of redundant condition code settings are unnecessary tests that precede branches 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. Consider the following productions that describe a ***tstl*** (comparison with zero) instruction on the VAX:

```
Branch↑'jle'↑previouscc → <=  Long↑a↑cca Long↑0↑previouscc
                                          Samecc↓previouscc↓'tstl'

Branch↑'jle'↑cctstl → <=  Longfafcca  Long↑0↑previouscc
                                          EMIT↓'tstl'↓a↓0↓0↑cctstl
```

Samecc evaluates to *true* if *previouscc* is same as the condition codes set by its second attribute, which in this case is the ***tstl*** instruction. Consequently, the test instruction is suppressed. If *Samecc* evaluates to *false* then a ***tstl*** instruction is emitted by the second production.

Inst ruction buffering and tracking

Attributes can be used to carry around arbitrary information to aid optimization. To provide additional power to the basic code generation process, these optimizations are incorporated in **a** cohesive manner as a *coroutine* to the parsing process within the attributed parsing **framework of** code generation. To realize these optimizations, instructions are buffered for the duration of a *basic block* [AHOU77]. Upon instruction selection, code is not directly emitted as output. Instead of a straight forward *match, generate, match, generate* code generation, many *matches* are'performed followed by a single *generate* at the end of a *basic block.* Instruction buffering facilitates local code motion, such as moving loads and stores closer to uses, remembering alias locations to eliminate redundant loads and stores and to use register addressing modes instead of memory locations. Furthermore, machine-specific idiosyncrasies such as register pairs, overlapped register use and the contents of the stack top are *tracked* by means of the instruction buffer. Within *basic blocks,* variables are kept in faster locations as much as possible. The code generator replaces references to the memory address of a variable by its equivalent register alias. The principles on which this strategy is based are:

- Whenever an assignment involves moving an operand from a cheaper addressing into a costlier one, the generation of the move instruction is delayed. Costs are defined in terms of storage classes such as memory, stack and register. For example, the cost of **a** register would usually be less than that of memory. Consequently, the store **of a register** into memory is delayed.

- Similarly, operand relocations may involve movement from a costlier addressing mode to a cheaper mode. An example of such an occurrence would be loading a memory variable into a register. In such cases, the move instruction is *hoisted* to the position after the last use of the cheaper addressing mode within the current basic block and after **all** intervening assignments to the memory variable. Then, all subsequent references to the memory variable are replaced by the register reference provided this replacement does not change instructions.

7. Grammar gyrations

The previous sections described and demonstrated the advantages of attributed parsing. The key point was the use of affix grammar machine descriptions to clarify many of the decisions in choosing code sequences. Lest the overall perspective be dimmed by this concentration of detail, we present a few questions that would probably have already arisen in the readers' mind:

How can other aspects of code generation, such as register assignment, be accommodated within the attributed parsing model? How do the grammar productions parse a superset of the IR? How to ensure that the attributed parser will not block? Is loop detection and elimination needed when attribute evaluation disambiguates parsing conflicts? How does having a restricted model for attribute evaluation affect the kind of code generated? Is multiplicity of grammar productions a potential problem? What grammar gyrations can reduce the total number of productions and consequently yield smaller code generator tables?

Register assignment

Invoking the action symbol *PEMP* is an *on-the-f/y* call for a temporary requirement. On irregular architectures, a value must be loaded into the right kind of register. For example, in a multiplication context, the operand location may need to be an *even-odd* register pair on the PDP-11 and the *AX-DX* accumulator pair on the iAPX-86. Thus, TEMP must be attributed by appropriate *storage class* requirements. The *inherited attributes* to TEMP may be the data type, machine storage class of the temporary location and any other specific requirement such as the need for the AX register on the iAPX-86. For example, TEMP↓Accumulator↓'word'↓AX denotes a specific requirement for the accumulator AX on the iAPX-86. This requirement is essential in cases of type conversion productions and for multiplication on the iAPX-86. Similarly, TEMP↓Even↓'word'↓ANY denotes **a** specific requirement for an even register on the PDP-11. In the absence of *inherited attributes* to LHS of productions, the context of operand use is determined by the prefix operator in the IR and the *left context* that will usually be available at a constant offset from the top of the parser stack. Thus, the affix grammar guides register assignment. However, there are no guarantees on conditions for register spills. In order to avoid register spills, a separate phase of global register allocation is required. Usually, global optimizers estimate (sometimes *guestimate)* register allocation based on frequency analysis of variables. The results of such estimations may be expressed as attributes to variables in the IR and eventually passed to TEMP during code generation.

Blocking

The language generated by the affix grammar for the target machine must be a superset of the IR. Otherwise, blocking due to mismatched IR specifications becomes a potential problem. For a correct IR of a given program, the code generator may block if the automaton has no valid next step and thus no match for the input IR. If the machine description is incomplete, **syntactic blocks** may occur. A **uniform** instruction set is defined as one in which operands of an operator are valid independent of context [GLAN77]. That is, uniform instruction sets allow any sequence of valid operands and operators via conversions or moves. If uniformity is established, **syntactic blocking** is impossible. Similarly, in a given configuration set, if all semantic qualifications imposed by attributes and predicates block production recognition then a **semantic block** may occur. The code generator attempts to match every production in succession but fails to match because all uses of the production pattern are restricted and the input IR does not satisfy these restrictions. To prevent semantic blocking at least one **default production** that has no qualification restrictions must be created. This production contains the syntax of the general form without semantic restrictions. When all preceding productions fail, this production will definitely match and consequently, it guarantees no blocking. Alternately, the user must establish that whenever a sequence of qualifications for different productions is considered, at least one will be satisfied. To predict potential blocking configurations, an analysis of the machine description is needed. The analysis highlights instances where non-orthogonality of the instruction-set can be hazardous to code generation. Depending on the context of such instances, the target-machine describer can decide, by hand, whether such a condition is acceptable as it is or whether it should be corrected by addition of extra grammar productions. For example, the non-existence of real operands as array indices may show up as an example of non-orthogonality. But, this condition is legitimate because it can be guaranteed that such a case would never appear in the input IR. Usually, syntactic blocks appear because a sufficient set of transfer productions are not supplied. Thus, when trying to convert an operand from one data type to another, the code generator reaches a state in which it cannot *shift* on the input. This problem is corrected by providing transfer productions between *every pair* of data types on the target machine. Potential code generator blocks can be automatically detected when a code generator is first created. An analysis of the code-generator automaton states is carried out to determine *uniform* occurrences of operators and operands. For each operator in the machine grammar, all its leftmost children **are** determined. As a sufficient condition to prevent blocking, anywhere an operator occurs in a machine description, all its leftmost children should be possible successors. The rightmost children of binary operators and other children of n-ary operators are analyzed similarly. Alternatively, the shift-reduce table output by the code-generator automaton can be inspected for completeness in actions for all operators and operands. If a state S is entered by *shifting* an operator *op* and there exists an action for that state and the next symbol *n,* then every state entered by *shifting op* must have an action for symbol *n.* Rightmost and other operands of the same operator are treated similarly. In an architecture, with addressing modes that are non-orthogonal to op-codes, if every legitimate addressing-mode/op-code combination is explicitly specified as a separate grammar production, then non-uniformity occurs due to the unspecified (and non-existent) combinations. In an affix grammar machine description, addressing modes are specified as *semantic attributes* in instruction-selection productions. Furthermore, the grammar being *type sensitive,* all op-code/addressing-mode combinations are taken care of by the very nature of the grammar specification mechanism. Non-orthogonal restrictions, if any, are present as predicates to productions. Thus, this kind of syntactic blocking is treated in a uniform way as semantic blocking. Another kind of syntactic blocking appears in certain architectures when data-types are non-orthogonal with respect to op-codes. For example, comparison operators may only work on integer data types only. Consequently, in a *type insensitive* grammar, comparison of real operands would be a syntactic block. In a *type sensitive* grammar, transfer productions could automatically convert real operands to integers and thus the syntactic block disappears However, for a more efficient representation of real comparisons, extra grammar productions could be specified with sequences of target code that implement real comparisons. In effect, such productions augment the instruction set of the target architecture.

The incremental inclusion of optimization productions may contribute to fresh *shift-reduce* conflicts in the machine grammar. The code generator would choose to *shift* in hopes of matching the longer production. However, at a later stage, the semantic restrictions on the applicability of this production may not be satisfied. Thus, the code generator may block if *bail out* productions are not supplied. The sequence of code that must be emitted on selection of a *bail out* production is usually determined by the grammar writer. However, it may be automatically obtained by removing the optimization and *bail out* productions and feeding a semantically valid but blocking IR input to the code generator. This IR would be parsed by other productions, yielding the relevant code sequence. For example, consider the VAX-l 1 grammar production describing an *sobgtr* instruction. The following *bail out* productions are needed to prevent blocking. A preprocessor can automatically include such productions in the machine grammar:

```
Long↑a↑ccdec →: =  Long↑a↑cc1 · Long↑a↑cc2 Long↑1↑cc3 > Long↑a↑cc4 Long↑0↑cc5 Label↑n
                                 ¬ Shortdistance↓n
                                 EMIT↓'decl'↓a↓0↓0↑ccdec
                                 EMIT↓'jgtr'↓n↓0↓0↑ccnotset

Branch↑'jgtr'↑'bgtr'↑cctstl →: =  Longfafccl · Longfafcc2 Long↑1 ↑cc3
                                 > Long↑b↑cc4 Long↑0↑cc5 Label↑n
                                 EMIT↓'decl'↓a↓0↓0↑ccdec
                                 EMIT↓'tstl'↓b↓0↓0↑cctstl

' Branch↑'jgtr'↑'bgtr'↑cccmpl → : =  Long↑a↑cc1 · Long↑a↑cc2 Long↑1↑cc3
                                 > Long↑b↑cc4 Long↑c↑cc5 Label↑n ¬ Zero↓c
                                 EMIT↓'decl'↓a↓0↓0↑ccdec
                                 EMIT↓'cmpl'↓b↓c↓0↑cccmpl
```

Looping

Another possible problem is **looping** in which the code generator continues to make moves, and possibly generate code, without consuming all the intermediate form symbols. This problem can only occur if an infinite chain of operand move productions is recognized, and, using qualifications, it can easily be avoided. In a *type sensitive* grammar, such loops usually occur when data type conversions are done more than once for the same variable. For example, a *byte* variable is converted to a word variable and then again converted back to a *byte* variable by a sequence of reductions without consuming any IR input. Thus, the reductions do not prune any of the IR and they return to the same grammar symbol. In such cases, restrictions are placed on predicates to ensure that conversions are performed only once for a given variable unless further IR input is consumed. An analysis of of grammar states, removing all predicates, is carried out and potential looping configurations are determined by tracing the use of replacement chains as described in [GLAN77]. Then, predicates are incorporated and restricted to avoid looping configurations.

Factoring

The machine grammar may be factored in a variety of ways as enumerated below. In most cases, factoring facilitates machine description and yields a smaller number of grammar productions. Consequently, the table size of a code generator is reduced. However, care must be taken not to factor the grammar at the expense of *readability* and *retargetability.* In particular, predicates may be shared by different productions to minimize code and consequently grammar factoring can be used to avoid replication of code. However, this technique will reduce the retargetability of the code generator.

- Simple uses of attributes allow the description writer to factor the machine grammar. Most importantly, addressing modes are factored out from instruction descriptions by the use of *addressing mode productions.* In effect, attributes are used to consolidate information about a reduced subtree.

- Predicates may be combined to factor symmetry in instruction specification. If two grammar productions have identical LHS and RHSs, with the only difference being their predicates, then only one production need be specified. This production represents the combination of these two productions and its predicate is the *disjunction* of the individual predicates. Thus,

  LHS $\rightarrow$ symbols Predicate, action

  LHS $\rightarrow$ symbols Predicate_2 action

  may be combined to form

  LHS $\rightarrow$ symbols Predicate, $\vee$ Predicate_2 action

- Operator factoring is possible but is not advised. In our experience, it becomes hard to debug the specification at the level of the grammar. For example, operator factoring creates problems in specifying certain attributes to operators (e.g., unsigned multiplication, time versus space optimizations when they conflict).

- Further grammar optimizations can be done by moving certain action symbols such as TEMP (i.e., calling the temporary assigner) out of *instruction selection* productions and representing it as a separate $\in$ production for an overall gain in clarity. In the absence of inherited attributes to LHS of productions, the relevant context (e.g., data type and storage class of the temporary) is represented syntactically in the LHS of $\in$ productions. Thus, TEMP↓'byte' and TEMP↓'word' would be represented as two distinct $\in$ productions:

Tempbyte↑t → ∈
TEMP↓'byte'↑t

Tempwordft → ∈
TEMP↓'word'↑t

Factoring action symbols improves the clarity of the machine grammar but increases the number of grammar productions and consequently the table size. Furthermore, care must be taken that the introduction of such ∈ productions does not introduce additional *shift-reduce* conflicts. In our experience, such conflicts did not appear.

8. Conclusions

Implementations of code generators (CG) for the PDP-11, VAX-l 1, iAPX-86, Z-8000 and IBM-370 exist. Code generators occupy roughly 100 Kbytes, generating about 50 lines of assembler code per second (real time). YACC [JOHN75] was used to generate tables for these implementations YACC's driver was modified to accommodate predicates and create parsers for affix grammars [FISC83]. It requires about four minutes to process each of the highly optimizing code-generation grammars containing about 600 grammar productions and 1200 parser states for rich instruction-set architectures such as the VAX-l 1 and very non-orthogonal instruction-set architectures such as the iAPX-86 and Z-8000. The PDP-11 and IBM-370 grammars contain about 400 grammar productions and 800 parser states CG produces good code when compared to native compilers on target machines and hand-written assembly code. The target code quality is comparable to the Portable C Compiler [JOHN78] with its post pass of peephole optimization and Intel's Pascal compiler for the iAPX-86. Most of CG's strength lies in using special machine instructions and subsumption of arithmetic within hardware addressing modes. Typically, one man month is needed to produce a code generator for a new target architecture. This duration may increase by another man week depending on the richness of the target architecture's instruction-set, such as the VAX-1 1, or the non-orthogonality of the instruction-set architecture, such as the iAPX-86 and the Z-8000. Much of the savings in time comes from the orthogonality of the affix-grammar productions. For example, all addresses are handled by a single set of productions. When constructing other parts of the specification, this information need not be continually borne in mind. Furthermore, separating addressing mode descriptions as grammar productions facilitates various addressing optimizations. One of the major problems of Graham-Glanville machine descriptions is the multiplicity of grammar productions" Since each op-code/data-type/addressing-mode must be written as a separate production, the grammar can easily get very large (about 8 million productions for the VAX). To produce a workable code generator, grammar factoring is essential. In spite of factoring, the total number of productions are substantial (over 1000 productions for the VAX) [GRAH82]. In our scheme, the use of attribute predicates and semantic treatment of attributes effectively replaces the cross product of op-codes and addressing modes by their additive sum. In effect, the addressing modes are factored from the instructions. This separation significantly reduces the total number of productions and consequently yields smaller code generator tables.

The use of semantic attributes provides for complex information flow, as well as selective rejection of inappropriate productions. Multiple instruction results, such as condition codes and auto-increment/auto-decrement side effects on registers, can be tracked as semantic attributes. Important target machine restrictions on the programming model such as:

- register restrictions for indexing, multiplication and division,
- segmentation restrictions on compilation (e.g., iAPX-86, Z-8001),
- data type restrictions for operands to instructions,
- addressing-mode use and access restrictions,
- irregular and inflexible register architecture (e.g., iAPX-86) and
- short displacements for branch restrictions

are conveniently handled by attributes and predicates that serve to band-aid such architectural restrictions. Thus, an architecture's weakness is made transparent to the compiler writer. On the other hand, an architecture's richness such as special addressing modes and fancy instructions are also captured in affix grammar productions using semantic attributes and predicates. The use of semantic attributes, attribute predicates and action symbols not only allows more information to be used to drive the parse, but also brings many of the decisions out of semantic routines into the grammar for an overall gain in retargetability. The semantic evaluation framework facilitates many optimizations paying more attention to the incremental development and improvement of a code generator. The main difference between simple code generators, and more elaborate ones, is how detailed the attributed descriptions are. Simple ones, that ignore *fancy* or obscure features can be readily produced. For example, arithmetic shifts may be ignored and multiplys or divides generated in all cases. Later, as time permits, and the need arises, more elaborate attributed descriptions, that more fully describe the target machine can be created. This approach allows an incremental development of a code generator, rather than an *all or* nothing approach.

In our scheme, adherence to single-pass code generation ruled out iterative machine-dependent and peephole optimizations. We are caught in a nasty cleft between the resulting optimizations gained on the one hand and speed and size of the implementation on the other. Many problems involved in generating high quality, optimized code remain open. Our code generation algorithm uses a restricted attributed parsing technique, forbidding inherited attributes to an LHS symbol. In some contexts, this attribute flow is inadequate. Thus, richer attribute flows must be investigated. The standard model of attribute evaluation assumes that a given attribute is evaluated exactly once. In certain aspects of code generation, most notably register assignment [LEVE82] and iterative peephole optimizations [WULF75], it is useful to change decisions that have already been made.

Thus, the applicability of *time varying* attributes in affix grammars and multi-pass attribute evaluation paradigms to code generation should be studied.

Acknowledgements

We are grateful to Bruce Carneal, Ed Desautels, Ray Finkel, Jack Fishburn, Chris Fraser, Sue Graham, Johannes Heigert, John Hennessy, Will Leland, Jim Morris, Don Neuhengen, Steve Scalpone, Keith Thompson and Bill Wulf for many favors We thank the referees for their extensive comments and invaluable help.

9. References

[AHOU73] A.V. Aho and J.D. Ullman, "The Theory of Parsing, Translation and Compiling", Vols. 1 and 2, Prentice-Hall, Inc., 1973.

[AHOU77] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley publishing co., 1977.

[CATT80] 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.

[CHAI82] G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring", Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Boston, Massachusetts, June 23 - 25 1982 (ACM SIGPLAN Notices Vol. 17 No. 6, June 1982).

[CROW82] T.R. Crowley, "Combining Table-Driven Effect Selection and Description-Driven Peephole Optimization for Automatic Code Generation", MS thesis, M.I.T. and Technical Report, Bell Laboratories, Department 45412, September 1982.

[DAVI80] J.W. Davidson and C.W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer", ACM Transactions on Programming Languages and Systems, Vol. 2 No. 2, 1980.

[FISC83] C.N. Fischer, M. Ganapathi and R.J. LeBlanc, "A Simple and Practical Implementation of Predicates in Context-Free Parsers", Computer Sciences Technical Report # 493, University of Wisconsin-Madison, April 1983.

[FRAS77] C.W. Fraser, "Automatic Generation of Code Generators", PhD thesis, Computer Science Department, Yale University, New Haven, Conn., 1977.

[GANA80] M. Ganapathi, "Retargetable Code Generation and Optimization Using Attribute Grammars", PhD dissertation and Computer Sciences Technical Report # 406, University of Wisconsin - Madison, 1980.

[GIEG83] R. Giegerich, "Automatic Generation of Machine Specific Code Optimizers", ACM TOPLAS July 1983.

[GLAN77] R.S. Glanville, "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers", PhD thesis, University of California, Berkeley, December 1977.

[GLAN78] R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation", Conference Record Fifth ACM Symposium Principles of Programming Languages, January 1978.

[GRAH82] S.L. Graham, R.R. Henry and R.A. Schulman, "An Experiment in Table Driven Code Generation", Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Boston, Massachusetts, June 23 - 25 1982 (ACM SIGPLAN Notices Vol. 17 No. 6, June 1982).

[JOHN75] S.C. Johnson, "YACC - Yet Another Compiler Compiler", Computer Sciences Technical Report # 32, Bell Telephone Laboratories, Murray Hill, New Jersey, 1975.

[JOHN78] S.C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symposium Principles of Programming Languages, pp. 97- 104, January 1978.

[KOST71] C.H.A. Koster, "Affix Grammars", in J.E.L. Peck (editor), ALGOL 68 Implementation, North Holland, 1971.

[LEVE81] B.W. Leverett, "Register Allocation in Optimizing Compilers", Ph.D. thesis and Technical Report CMU CS-81-103, February 1981.

[LEVE82] B.W. Leverett, "Topics in Code Generation and Register Allocation", Technical Report CMU CS-82- 130, July 1982.

[MCKE70] W.M. Mckeeman, "Peephole Optimization", CACM Vol. 8 No. 7 pp. 443-444, 1970.

[POPL82] M. Ganapathi and C.N. Fischer, "Description-Driven Code Generation Using Attribute Grammars", Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 25 - 27, 1982.

[PURD80] P. Purdom and C.A. Brown, "Semantic Routines and LR(k) Parsers", Acta Informatica Vol. 14 Fasc. 4, pp. 299-316, 1980.

[RUDM79] A. Rudmik and E.S. Lee, "Compiler Design for Efficient Code Generation and Program Optimization", ACM SIGPLAN Symposium on Compiler Construction, Colorado, August 1979.

[SIGN83a] M. Ganapathi and C.N. Fischer, "Automatic Compiler Code Generation and Reusable Machine-Dependent Optimization -- A Revised Bibliography", ACM SIGPLAN Notices Vol. 18 No. 4, pp. 27 - 34, April 1983.

[SIGN83b] F.C. Chow and M. Ganapathi, "Intermediate Languages in Compiler Construction -- A Bibliography", ACM SIGPLAN Notices Vol. 18 No. 11, November 1983.

[SPE84] M. Ganapathi and C.N. Fischer, "Attributed Linear Intermediate Representations for Retargetable Code Generators", Software - Practice and Experience, *to appear April,* 1984.

[STAN84] M. Ganapathi, J.L. Hennessy and V. Sarkar "Reverse Synthesis Compilation for

Architectural Research", Technical Report # 257, Computer Systems Laboratory, Stanford University, March 1984.

[SURV82] M. Ganapathi, C.N. Fischer and J.L. Hennessy "Retargetable Compiler Code Generation", ACM Computing Surveys' December 1982.

[SURV83] M. Ganapathi, J.L. Hennessy and C.N. Fischer "Retargetable Code Generators", ACM Computing Surveys, Surveyors Forum, Vol. 15, No. 3, September 1983.

[TANN82] AS. Tannenbaum, H. vanStaveren and J.W. Stevenson, "Using Peephole Optimization on Intermediate Code", ACM TOPLAS Vol. 4 Number 1, pp. 21 - 36, January 1982.

[WATT77] D.A. Watt, "The Parsing Problem for Affix Grammars", Acta Informatica 8, Springer Verlag , 1977.

[WULF75] W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs and C.M. Geschke, "The Design of an Optimizing Compiler"' American Elsevier Publishing Co., 1975.

[WULF80] W. Wulf, B.W. Leverett, R.G.G. Cattell, SO. Hobbs, J.M. Newcomer, A.H. Reiner and B.R. Schatz, "An Overview of the Production-Quality Compiler-Compiler Project", IEEE Computer Vol. 13 No. 8 pp. 38-49, August 1980.