Reverse Synthesis Compilation for Architectural Research

Mahadevan Ganapathi
John Hennessy
Vivek Sarkar

Technical Report: CSL-TR-84-257

March 1984

This project has been supported by IBM under contract 40-81, and by the Defense Advanced Research Projects Agency under contract MDA 903-83-C-0335.

Reverse Synthesis Compilation for Architectural Research

Mahadevan Ganapathi
John Hennessy
Vivek Sarkar

Technical Report NO. 84-257

March 1984

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

Abstract

This paper discusses the development of compilation strategies for DEL architectures and tools to assist in the evaluation of their efficiency. Compilation is divided into a series of independent simpler problems. To explore optimization of code for DEL compilers, two intermediate representations are employed. One of these representations is at a lower level than target machine instructions. Machine-independent optimization is performed on this intermediate representation. The other intermediate representation has been specifically designed for *compiler retargetability*. It is at a higher level than the target machine. Target code generation is performed by reverse *synthesis* followed by *attributed parsing*. This technique demonstrates the feasibility of using automated table-driven code generation techniques for *inflexible* architectures.

Key Words and Phrases: Compiler design, Retargetability, Intermediate Representation, Code-Generator-Generator, Code-Generation, Optimization, Reverse Synthesis.

Table of Contents

1. Introduction

1.1 DEL Architectures

Computer architectures other than the traditional general register machine have the potential for improved ratio of performance/cost. Oce particularly interesting class of traditional architectures arc the Direct Execution Language (DEL) machines [Flynn 80, 83]. They form a class of von-Neumann architectures that attempt to make the execution architecture suitable for a high level language (HLL). DEL architectures use built-in knowledge of the language environment to add more function in hardware to reduce system costs. The main thrust of this approach is to remove register and resource allocation problems that exist in familiar machines and also to avoid problems exacerbated by limited size of storage. DEL machine architectures are determined, in part, by the demands of programming languages. There is a one-to-one correspondence between functional operations in the HLL and instructions specified in the target architecture. Each action in the HLL evokes exactly one instruction in the image processor. Consequently, there arc no overhead instructions, such as *load/store* in the target instruction-set. DEL identifiers correspond to HLL variables rather than host registers or storage cells. There is a one-to-one correspondence between objects named in the HLL program and operands specified in the image architecture. Thus, the flexibility of the target machine is sacrificed to decrease the *semantic gap* between it and the HLL. In effect, DELs are interpretive architectures. Consequently, DEL machines may be too in flexible for a general purpose environment. *A general-purpose* DEL aims for encoding efficiency (both static and dynamic). The microarchitecture is optimized for code compaction. DEL representations minimize the number of bits needed in the instruction stream for operand specification, without resorting to frequency-of-occurrence based encodings. The instruction format is designed for an absolute minimum number of bits for each instruction in each procedure. In effect, the instructions are *bit aligned.* The opcode appears last and operand references are variable in size and format. The instruction fields are encoded using $\lceil \log_2 n$ bits, where n is the number of possible identifiers in the HLL. Instructions locate operands using a table, called the *contour memory.* The contour table contains the value for simple constants, variables and labels or address for structures and non-local variables. This information is kept for each object named in the HLL program.

DEL architectures can potentially outclass traditional register architectures in size *of static code* and *data-traffic bandwidth.* The simplicity of addressing structure permits simple direct addressing and the use of immediate operands rather than complex effective-address gcncration. Studies in DEL architectures have shown that the number of instructions required to execute programs can be reduced by a factor of three when compared to traditional architectures [Wakefield 83]. A *contour memory buffer* in a DEL microprocessor can reduce on-chip memory arca and enhance off-chip memory bandwidth. However, the execution-time

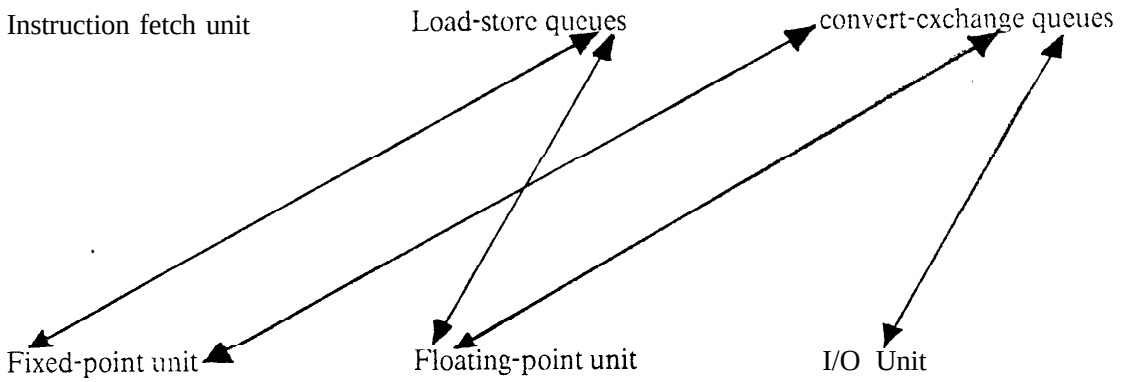
performance of such machines may-be a potential problem in a general purpose environment. Since instructions arc not aligned, instruction fields need to be *extracied.* Consequently, the instruction-interpretation pipeline must bc handled carefully,[1] otherwise it cannot be very deep.

DEL design represents a series of tradeoffs in architecture design; the most important tradeoff being space savings duc to conciseness in representation versus interpretation time. There exists a conflict between efficient representation of a HLL program and efficient interpretation by a host system. At the one extreme, direct interpretation without the use of a compiler is possible, but such an approach has limited suitability to many traditional languages. At the other extreme, most of the translation can be accomplished by compilation to microinstructions followed by minimal interpretation. Tradeoffs in the above conflicting demand have a strong bearing on the interface between compilation and interpretation. A cost-effective design and implementation of a DEL is still the subject of on-going research. Studies of alternative DEL architectural strategies requires close cooperation between compiler writers (so that the machine has an adequate set ofprimitives) and hardware designers. Measures of architectural efficiencies must bc done by comparing the results for DEL machines with the results for traditionally organized machines. This comparison has not been possible, since both DEL compilation and DEL architectu es have not received anywhere near the study that compilation for traditional architectures has received.

1.2 FOM

FOh4 is a Forcran Optimized Machine [Brantley 82, 83] that has been designed with the motivation for high performance in scientific computations. It is a *high performance* DEL rather than a *general purpose* DEL. In a general purpose DEL, state transitions of both image and host occur in the same order, thus, preserving the *transparency* of representation. Multiple image instructions cannot be simultaneously executed and consequently, the architecture cannot bc pipelined. However, contingent on the interpretation order being the same as the image sequence, overlapped execution is allowed. To gain overall execution speed, FOM violates *transparency* and allows code scheduling. It trades execution speed for code compaction and has fixed size instructions that can be extracted in at most one pipe cycle. FOM's CPU has multiple functional units: Fixed Point Unit to operate on integer, boolean, logical, label and array-base data, Floating Point Unit, Load/Store Unit to load and store array clcmcnts, Exchange/Conversion Unit to move data between units, Instruction Unit to fetch and decode, and an I/O Unit for formated I/O. The following diagram depicts this organization.

'Memory may bc bit-packed. but the Instruction-Cache need not bc bit-packed.



A number of tradeoffs were made between the organization and the instruction-set architecture of FOM. FOM does not have condition-codes. Being a shared resource for multiple units, condition -codes degrade performance. Consequently, boolean operations produce explicit results rather than set condition-codes and are three-address instructions. Jnstead of general purpose registers, FOM provides a special hardware-managed buffer called the Access Look Aside Table (ALAT). This buffer is a large register-set that is automatically block-loaded on proccdurc entry. There are four sets of registers and their block-loading can be overlapped with the execution of the current procedure by suitable scheduling of *NxtProc* instructions. Auto-indirection is another feature of this buffer that tends to produce compact code for accessing procedure parameters and global variables. The auto-indirect bit of an ALAT entry indicates whether the data field contains the data or the memory address of the data. Each operand reference to an auto-indirect ALAT location causes a memory reference. The set of variables accessible by a procedure is recorded in memory and in the Access Table (AT). The Environment Pointer (EP) points to the AT of the current procedure. While a proccdurc is executing, information in the next procedure's AT is moved into the ALAT. ALAT is divided between two units according to the data type: ALATI and ALATR. Thus, the high-speed buffer can hold 128 fixed-point and 128 floating-point entries in which all local variables arc automatically loaded. Since the ALAT is 4-way interleaved, there exists a need to avoid ALAT bank conflicts. An expression-evaluation stack is used to reduce the number of rcfcrcnces to ALAT.

The FOM instruction-set architecture was designed with the following motives:

- high static coding efficiency for Fortran programs -- reduced number of instructions per Fortran instruction. Examples of high-function operations in FOM arc *NxtProc*, *Autoindirection* and formated I/O.

- simplicity of implementations to improve cycles ptr instruction -- high performance with multiple

functional units. A separate floating-point unit, load/store unit and conversion unit enhance performance.

- increased cxccu tion overlap -- delayed branches and communication between units via architccted FIFO queues.

While DEL architectures have the potential benefit of requiring fewer number of instructions to encode a MLL, appropriate compiler technology must be employed to take advantage of this benefit. This paper develops compilation strategies for DEL machines and tools to assist in the evaluation of their efficiency. In particular, we describe the implementation of Fortran and Pascal compilers for FOM. This exercise is interesting for the following reasons:

- The effect of machine-independent optimization is determined for DEL, architectures.
- Code generation is a *reverse synthesis*[2] issue and a pattern-matching code generator is used to efficiently *compile back* the reverse synthesized code.
- It demonstrates the use of a table-driven code generator for purposes of architect1.ural research in the environment of a changing instruction set and optimizations.

1.3 Compilation Strategy

The primary purpose of this exercise is to develop approaches to generating machine code for a variety of different language-oriented architectures as well as traditionally organized machines. Some of the new machines may require substantial departures from traditional compilation techniques. Since there is an exact mapping between the execution architecture and its host HLL, compilation is expected to bc straightforward for the host HLL only. To host a foreign HLL or to adapt to changing instruction-sets of the DEL itself, a rctargctable code generation scheme is required. Moreover, much of DEL research has ignored code optimization. For maximum effect of optimization and for centralizing a global optimizer, such optimization is traditionally performed on low-level intermediate code. In our scheme, this optimization is performed on U-code. Conscqucntly, code generation for DEL machines becomes *a reverse synthesis* process.

Furthermore, our aim is to support architectural experimentation at the level of determining what the impact of a particular architecture is on the aggregate code size and execution time for a particular system. To make such comparisons it is necessary to have compilers for the different architectures. A fair comparison can best be achieved by using a single front-end with several different back-ends; one for each architecture of interest. By using a single front-end, many of the differences that exist between compilers for different machines will bc climinatcd. However, construction of a new back-end for cach architecture of interest encounters two major problems:

[2] Section 1.5 discusses reverse synthesis.

- It is extremely costly: each back-end can require several man years of effort.
- It distorts the study, because of the differences that are introduced in the back-ends and the relative abilities of the compiler writers.

As an alternative. we have employed a table-driven code generator. For general background material on *automated retargetable code generation*, the reader is referred to a bibliography [Notices 83a] and a survey and critique of code generation models [Surveys 82, 83]. In our code generation schcmc, the code generator is organized as a parser and the machine description plays the role of a grammar. Predicates [Fischer 83] are used to extend the range and scope of context free grammars to produce code generators. Code generation is performed by *attributed parsing* [Popl 82, Toplas 84].

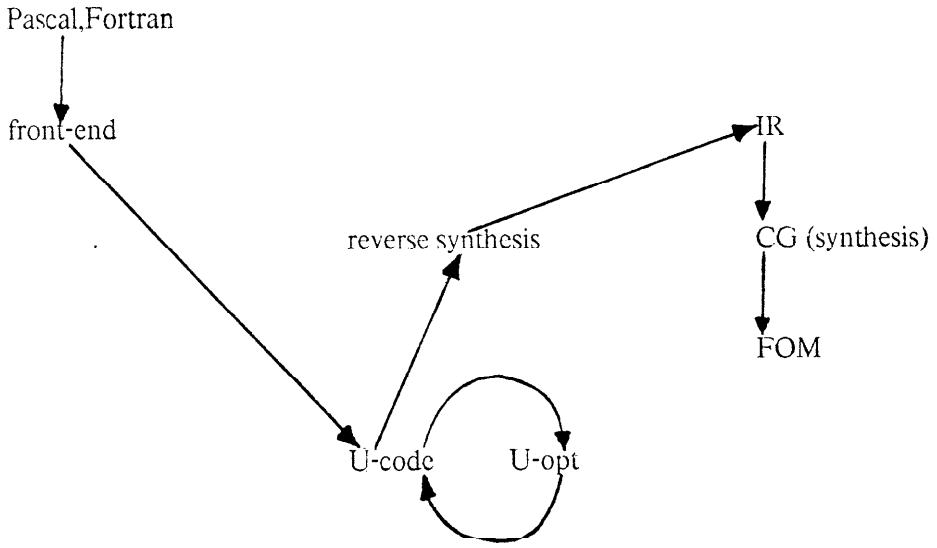
Compilers for FOM are organized as follows.

- A front-end translates the programming language (e.g., Fortran, Pascal) to an intermediate representation called U-code [82].[3] This low-level intermediate form allows for representing more details of program code.
- A global optimizer is used to perform a variety of machine-independent optimizations at the Ucode-level [U-opt 83]. The optimizer is centralized at this level since it can see (and thus optimize) more details.
- The reverse-synthesis phase translates U-code to an *attributed prefix linear intermediate representation* called IR [Uwtr 81, SP&E 84].[4] IR has been specifically designed for retargetable code generation.
- The instruction selection phase translates IR to FOM assembler code [Popl 82]. The code generator is automatically derived from a code-generator-generator (CGG).

---

[3] Pragmatically, U-code was selected because there is an optimizer available on U-code. Otherwise, front-ends may generate IR directly.

[4] [Notices-83b] contains a bibliography on intermediate representations.



Following sections describe each of these phases in more detail.

Our effort parallels the PL.8 [82] effort; first, the HLL is translated to the instruction set of a simple abstract machine that is at a *lower level* than the target CPU. Then, machine-independent optimization is performed on this low-level intermediate representation that is partially machine-dependent. Optimization is partitioned into many independent operations that are independent of particular source-language constructs. The intermediate program is translated to an equivalent program with reduced running time. Important differences between our approach and the PL.8 approach are:

- In the FL.8 compiler, register allocation drives code selection. All computation is performed in registers and there is heavy dependence on a global register allocator. Furthermore some optimizations may conflict with the register allocator (e.g., packing versus increased life times for variables). On machines that do not have any registers (e.g., FOM) or on machines with a very large number of registers, this compiler organization may be inappropriate. Thus, in our organization register allocation does not drive code selection.

- We do not perform trap elimination and elimination of unnecessary branches, code straightening and reassociation.

- Our instruction selection algorithm is table-driven and employs a pattern-matching approach. This technique is used to efficiently "compile back" a sequence of low-level intermediate operations to single target machine operations. The PL.8 compiler does only a limited amount of this optimization.

1.4 Front-end and Global Optimizer

The front-ends for Pascal (UPAS) and Fortran (UFORT) translate an extended version of standard Pascal and Fortran-66 to U-Code. U-Code is largely machine-independent, so the bulk of the front-ends are not affected by the differing target machines. The only machine dependency in U-Code (ignoring register allocation which is not relevant in the FOM code generators) is the semi-symbolic storage allocation. This machine dependency is included for two reasons:

1. To allow the expression of addressing calculations that may be optimized by the optimizer.

2. To permit the straightforward expression of detailed storage allocation that may result from storage packing. U-Code uses bit-addressing to express low-level storage allocation details; without bit addressing, highly machine dependent packing and unpacking routines would be required.

The storage allocation strategy for each target machine is encapsulated in a set of about twenty constants that dictate the packed and unpacked storage sizes for all the primitive data types, the addressability guidelines, and the alignment restrictions. Generating a front-end for a new target machine usually takes less than a day.

The optimization phase uses two key programs:

- PMERGE - an in-line procedure expander that does selective in-lining of procedures based on time-space tradeoffs. Although this optimization can lower calling overhead, its major role is in lengthening the code segments between procedure calls to increase the benefit to be obtained by register allocation. The merger is run prior to the main optimizer.

- UOPT - a U-Code to U-Code global optimizer, described in detail in [U-opt 83].

The main optimizations performed by UOPT include:

1. A large set of local optimizations: common subexpression elimination, stack height reduction, constant folding, etc.

2. Loop optimizations: strength reduction, induction variable elimination, forward and backward code motion.

3. Global optimizations: constant propagation, common subexpression elimination, dead store and code removal.

4. Global register allocation: using priority based coloring.

The U-Code representation is complete and no additional information is required by UOPT. It uses the method of Partial Redundancy Elimination as its main optimization technique. Using the entire optimizer with the merger, performance improvements of 45-65 percent are achievable. Much of this benefit comes from optimizing array indexing and addressing calculations.

1.5 Reverse Synthesis

The motivation for this phase arose from observing that *optimizability* and *retargetability* can be conflicting requirements in an intermediate language. A *lower* (i.e., closer to machine) level intermediate language can represent more details suitable for optimization; a *higher* level intermediate language can be targeted to a wider range of machines. We found this to be true in the two intermediate languages, U-code and IR. U-code is at a lower level, in that it is defined in terms of a hypothetical stack machine. As described in the previous section, we have front-ends and a sophisticated global optimizer for U-code. IR, however, is closer to a HLL than the machine level. IR is designed for automated retargetable code generation, using attributed parsing. We decided to develop Fortran and Pascal compilers for FOM by using the front-ends and optimizer from the U-code system, and the table-driven code generator from IR. A translator from U-code to IR was thus required, and it is this translation from a lower level to a higher level that we call *reverse synthesis.*

U-code and IR

U-code and IR are similar, in that they are both designed to serve as intermediate languages. Both of them are *HLL-independent* (i.e., the same code generator can support intermediate code obtained from different HLLs), as well as *machine-independent* (the same front-end can generate intermediate code to be used by different machines). Both U-code and IR have the notion *of data types.* They also share the notion of *storage classes,* which are used to describe the kind of memory a data object should belong to (e.g. constant, static, dynamic, register). The fundamental differences between U-code and IR are:

- U-code data objects are defined by locations in the hypothetical U-machine's memory areas, whereas IR has variables as in a HLL. Address arithmetic is more explicit in U-code, and therefore more optimizable. Also, it is easier in U-code to express those HLL features that are semantically defined at the machine level (e.g. packing,overlaying). Thus, we see that a lower level intermediate language, like U-code, can express a wider range of HLL features. On the other hand, a higher level intermediate language, like IR. can support a wider range of machine features. U-code is less targetable than IR because of its low level allocation of data objects in memory. FOM requires that fixed-point and floating-point objects be allocated separately. This separation is simple in IR, but impossible in U-code.[5]

- IR's syntax has been designed for table-driven code generation. One feature of IR's syntax is its *prefix* representation of expressions, which is well suited for bottom-up attributed parsing. U-code, instead has a *postfix* representation, since an expression is denoted by a sequence of stack operations.

- IR's design for automated code generation makes it more extensible than U-code. Modifications in IR can be easily implemented by modifying the formal specifications of its scanner and parser. Further, *attributes* provide flexibility in extending IR.

---

[5] If the target architecture's storage model does not conform to the U-code model, the symbol table is needed to reallocate data objects.

Method

The *reverse synthesis* from U-code to IR is done by a single-pass, syntax-driven translator. Conventional code generation from U-code to a machine level does not need to bc syntax-driven, as each U-code instruction is merely expanded into a sequence of machine instructions (i.e., a 1:n mapping). However, when translating U-code to IR, we encounter n:1 mappings, where a structured sequence of U-code instructions maps into a single IR statement. The process of identifying such a sequence of U-code instructions, is exactly that of *parsing* the U-code program.

The major issues encounter-cd in this translation are:

- Mapping U-code data objects to IR variables - In general, it is impossible to derive the structure of HLL variables from a U-code program. It can only be done in simple cases, like *scalar* variables. Hence, we used a *symbol table* to augment the input U-code program for the purpose of translating to IR. The *symbol table* is a well defined entity in the U-code system, as it is used by the debuggcr. It contains information about the type and structure of HLL variables and their mapping to U-code locations.

- Identifying temporaries - Not all U-code objects arc defined in the *symbol table.* In particular, *temporaries* are U-code data objects that do not correspond to any HLL variable. A U-code *temporary* can be easily identified by checking if the U-code location has an entry in the *symbol table* (the *symbol table* would have to keep track of the *range* of locations, for non-scalar objects). IR permits "on the fly" variable declarations, which can bc interspersed with IR statements, provided there is no forward reference. *Temporaries* can be declared in this fashion.[6]

- Conversion of address offsets - All U-code address arithmetic is done directly in bits. IR's address arithmetic is at a more abstract level; it uses *address-of* and size-ofoperators, like those in the programming language C. Thus, a U-code address is mapped to an address-ofoperation on the corresponding IR variable. A U-code bit offset is mapped to a size-ofopcration on the base type of the structure being accessed. The only problem occurs for *address* temporaries, where the base type is unknown. It can bc determined by using the base type of the address expression assigned to the temporary.

- Generating prefix code - The limited problem of translating U-code's *postfix* expressions to IR's *prefix* expressions appears to have a simple solution - just read the *postfix* expression backwards! However, this only works if all the operators are commutative, since a backwards scan reverses the order of operands. For that and many other reasons, it became necessary for the translator to build an *expression tree,* and gcncratc prefix code by a *prc-order* traversal.

---

[6] We do not need any symbol table information for deciaring temporaries because they are always scalar and their types are defined in the U-code instructions.

1.6 An Example

```
subroutine p(y,n)
common a,b,i,j
n = i
Y= a
return
end
```

The Fortran compiler front-end and reverse-synthesis phase produced the following IR. The symbol ↑ attaches an attribute to a variable. The order in which attributes of the same variable appear is unimportant. Note that a COMMON is split into separate fixed-point and floating-point arrays.

```
:p ↑2parameters      blockbegin

    : c$3 ↑Import
    : c$4 ↑Import
    : n ↑parameter ↑pointer ↑1unit
    : y ↑parameter ↑pointer ↑1unit

    : = INDIRECT↑integer + n * 0 size↑integer INDIRECT↑integer + * size↑integer 0 c$4

    "the code generator will constant-fold the multiplications"

    : = INDIRECT↑real + y * 0 size↑real INDIRECT↑real + * size↑real 0 c$3

    blockend

: c$4 ↑Global finteger ↑2units
: c$3 ↑Global ↑real ↑2units
```

2. U-code to IR translation

This section discusses in detail the reverse synthesis of IR from U-code.

2.1 Motivation

Existing *buck-ends* in the U-code system have been developed using UGEN[Ugen 82], a rctargetable code generator. The effort in UGEN is to separate the machine-independent and machine dependent modules of a code generator program. It is based on a structured programming approach, thus enabling different *buck-erzds* to use the same machine-independent modules. The *retargetability* of such a code generator arises from the fact that only the machine-dependent modules need to be modified, to develop a code generator for a new machine. The problem in UGEN lies in the assumptions it makes about the structure of a conventional machine that fail miserably when considering unconventional architectures, like FOM. Therefore, even the *machine-independent* modules of UGEN will have to bc modified for such architectures. Instruction selection, using UGEN's technique, poses some problems too. Duc to the interaction of all addressing modes and op-codes, there exists a typical n^2 problem. Thus, each individual instruction-selection case requires some thought. Furthermore, it is hard to *compile-back* complex instructions. In effect, the entire code generator has to be rewritten, as existing UGEN code generators cannot be rctargeted to DELs such as FOM.

An alternative to the hand-coded approach in UGEN, is to use a *table-driven* code generator.[7] *CG* [Ganapathi 80] is one such implementation that uses attribute grammars and a machine-independent/language-independent intermediate form, JR. Thus, a U-code to IR translator is needed to use CG with existing *front-ends* in the U-code system.

2.2 U-code vs. IR

Both U-code and IR arc designed to be machine-independent/language-independent intermediate forms. They differ greatly in syntactic appearance,, but that is not an issue. There also exist a few fundamental differences between the two languages, that are discussed in this section.

---

[7] [Surveys 82] provides an overview of such au tomalic code generation techniques that separate the machine description from the code generation algorithm.

2.2.1 U-machine memory vs. IR variables

U-code objects are defined in terms of a hypothetical "U-machine" that has six different memory areas:

1. Read-only code store (also holds string constants).
2. Expression stack
3. Static storage (S memory)
4. Heap
5. Registers (R memory)
6. Memory stack (M memory)

U-code instructions operate only on these memory areas, and U-code objects are defined by their locations in them. Since U-code provides no information about which memory locations correspond to different data objects, the *back-end* is forced to allocate each U-machine area indivisibly. This approach is restrictive, in that data objects are forced to be allocated in the same order as they occur in a U-machine area, the order enforced by the *front-end*. However it is possible that the target machine may prefer/require a different sequencing of objects, e.g., FOM requires that fixed point and floating point objects be allocated separately.

IR, instead has *variables* just as in a high-level language. The advantage in this approach is that IR makes no commitment about the actual sequencing of its objects in memory areas. This storage-binding problem is postponed to the *back-ertd* that is better equipped to handle it. In CG, the storage binding phase is driven by a description of the target machine's memory areas.

Each IR variable must be declared to belong to some *storage class*. Examples of IR storage classes are:

- ↑F - Function return value
- ↑G - Global
- ↑L - Local
- ↑M - Actual parameter
- ↑P - Formal parameter
- ↑T - Temporary

U-code also imposes a similar classification on its objects implicitly - such a classification could be deduced by an analysis of the U-code program, as is done by code generators in the U-code system.

2.2.2 Types in U-code and IR

Both U-code and IR are *typed* languages. This property implies that all objects, in both languages, must have an associated data type. Also, both languages have a similar type classification. The difference lies in the way types are *declared.*

U-code types are declared through U-code instructions that state the type of their operands. IR, instead, uses variable declarations to specify variables' types, as in a high-level language. Thus, IR operators may be *polymorphic*, in that the same operator symbol can be used for operations on different types. Consider addition as an example. In U-code, the ADD instruction has a field to identify it as fixed-point **(Data** type I, J or L), or floating point (Data type Q or R). In IR, the + operator is used in both cases. Since the data type of all variables are known from their declarations, the type of an operator is determined by the type of its operands.

Justification for the IR approach consists of:

1. Using the same IR operator for operations on different data types leads to smaller parse tables.
   An alternate solution could be to group all typed occurrences of an operator into a single terminal symbol, in what would be an *abstract syntax* of IR. The type may be considered an *attribute* of the operator that plays no role in the parsing of IR, but is available to the code generation routines.

2. Untyped operators allow for the possibility of "mixed mode" expressions like + real *integer.* This allowance is convenient for the *front-end*, since it can leave the *type coercion* to CG that can convert the *integer* **to** *real.*
   Type coercion could be a language-dependent issue that must be resolved by the *front-end*, rather than be incorporated in the intermediate form. It is possible that some language may define an unconventional *type coercion,* such as *integer := real + integer* should be performed by coercing the real to *integer,* rather than the *integer* to *real* and then the sum to *integer.* This enforcement would violate IR's automatic *type coercion* convention. To enforce the HLL convention, a *convert* operator is used in IR. The attribute of the convert operator specifies the type to which the operand is to be converted. The IR for the example would then read:

   : = Integer + Convert↑integer Real Integer

3. Thcrc exist some machines (e.g., *fagged* architectures), where the same operator (opcodc) is used for operations on different types, i.e., *Add real,integer* is valid. Typed operators in IR will not be ablc to support such operations.
   However, such architectures can also be accommodated in a typed operator scheme, by including *untyped* as a possible type for an operator.

Just as storage classes arc used in both U-codc and IR to provide information for the storage-binding phase of

code generation, types could be used to qualify operators and provide information for the *instruction-binding* (or *code selection*) phase. If this qualification is done in IR, it would still be necessary to retain types in variable declarations, because type information will be needed in the storage binding phase.

2.2.3 U-code stack operations vs. IR prefix expressions

U-code stack operations and IR prefix expressions are two diverse representations of expressions in an intermediate form.

U-code stack operations essentially form a *postfix* representation. The major advantage of a *postfix* expression is that it can easily be evaluated using a stack (which is exactly what the hypothetical U-machine has). Consequently, it is simple to develop an interpreter for such an intermediate form, and indeed the U-code system does have a U-code interpreter (UINT).

Some advantages of a *prefix* notation over a *postfix* notation are outlined in [Uwtr 81](pp. 14,15). The gist of their argument is that an operator establishes a certain context, in which its operands are to be interpreted. It is thus more convenient for a code generator to process an intermediate form in which the operator precedes its operands. A prefix notation has this property. This property is a definite advantage if a single pass code generation scheme is being used (as in CG). Naturally, it is of no consequence if, say, the *back-end* builds expression trees and uses tree pattern matching techniques to generate code.

2.2.4 Address arithmetic in U-code and IR

The degree to which address arithmetic is permitted in an intermediate form is a good indication of how much closer the intermediate form is to a low-level machine language, rather than a high-level language. If an intermediate form permits general address arithmetic (e.g., indexing to access the field of a Pascal record), it has already made some assumptions about the target machine's memory structure that may be too restrictive. Consider floating point double word alignment as an example. Some machines may not enforce this alignment, and may allow a floating point datum to be allocated at any location. Other machines that do enforce this alignment require that it be allocated on a double-word boundary. This requirement obviously affects address calculations, indicating that the same intermediate form program will not be targetable to these two different kinds of machines.

However, if an intermediate form does not allow address arithmetic, it must possess some primitive data structures that can implement high-level data structures in a machine-independent fashion. For example, IR has one-dimensional arrays that can also implement multi-dimensional arrays. The problem here is that the intermediate form may now become too HLL-dependent.

IR scores a major plus over U-code in address arithmetic due to a simple abstraction. The size of a primitive data type in IR is expressed by the keyword "size", attributed by the data type, e.g., size↑i specifics the size of an integer. In U-code, all address arithmetic is done in *bits*. The "machine independence" of U-code arises from the fact that a *front-end* can be trivially modified to generate U-code for a particular machine. The modification is in a table of constants that gives the actual bit size of each primitive type. A separate table is created for each possible target machine. Thus, a particular U-code program can only be translated to the target machine it was *intended* for, i.e., the target machine for which the *front-end* used the table of constants, while generating U-code.

2.2.5 Extensibility and Robustness

Perhaps the most important test of an intermediate language is how well it is able to accommodate new HLLs and new machines. The IR is bound to need modifications, because the new HLL/machine may have some novel features that were unheard of when the IR was designed. The ease with which these modifications can be implemented is a direct measure of the extensibility and robustness of the IR.

Herein lies the biggest difference between U-code and IR. Attributes make IR highly extensible; for instance, new types and new storage classes can be easily incorporated by adding new attributes. Further, the IR system uses *table-driven* code generation techniques. An advantageous consequence of this approach is that the *back-end*'s scanner and parser are both table-driven. Hence, a modification in IR can be supported by merely modifying the formal specifications of the scanner and parser, rather than modifying hand-written code in the *back-end*.

U-code does not have attributes, but its structure is uniform enough to facilitate extensibility. Implementing an extension is a different matter altogether, since the U-code system does not have table-driven scanners and parsers. A modification is implemented by changing the U-code Reader and U-code Writer programs, a task that is generally recognized as costly and cumbersome.

2.3 m:1, 1:m, 1:1 & m:n transformations

One method of comparing the expressive power of two programming languages, is to consider the transformation of actions or data declarations from one language to another. The number of primitive statements/instructions used to represent the same action/declaration in the two languages, gives a comparative measure of the expressive power of the two languages. The language that uses fewer statements/instructions is more powerful. If we consider transformations from language A to language B, the following cases arise:

- m:1 - A needs more than one statement to express a construct that can be written in B using exactly one statement.
- 1:m - The converse; one statement in A and many statements in B.
- 1:1 - The construct can bc expressed by exactly one statement in both A and B.
- m:n - The construct needs more than one statement in both languages.

Comparing U-code and IR in this respect, WC see that most constructs lead to m:l transformations, implying that IR is more expressive than U-code.

2.4 Examples

In this section we provide a few examples to informally illustrate the issues involved in translating from U-code to TR. Both the U-code and IR fragments presented have been generated by programs - the U-code by the *front-end*, and IR by the translator. In some cases they have been annotated and indented to improve readability.

2.4.1 IR Declarations

As described in Section 2.2.1, IR has variables, whereas U-code does not. For practical purposes (described in Section 1.5), WC augment the input U-code program by a *symbol table* that can be optionally generated by all the *front-ends* in the U-code system. The symbol table format is described in [U-code 82]. This use makes our translator strictly a "(U-code + Symbol Table) to IR translator", rather than just a "U-code to IR translator". In fact, there arc no U-code instructions that correspond to IR's variable declarations except for PSTR that defines a formal parameter.

Using the symbol table leads to IR variables having the same names as the corresponding HLL variables. This property serves as a very helpful development/debugging aid in the design of the translator. An example of Fortran variable declarations follows:

Fortran Declarations:

```
INTEGER A, B(lO)
REAL X(5), Y
```

U-code symbol table:

```
% $MAINB 1 1
V $ A     S 1 128 <1 J>
V $ B  S  1 160
```

```
<4 A 32 <5 U 110 <1><1>
V $ x     S 2 0
<4 A 32 <5 U 15 <1><2 R> >
v $ Y     S 2 160 <2>
#
```

Generated IR declarations (comments written in "s):

```
: A ↑L↑i↑1 " ↑L = Local, ↑i = integer"
: B ↑L↑i↑10 " ↑10 = size "
: X ↑L↑r↑5
: Y ↑L↑r↑1
```

Here is another example with Pascal records:

Pascal declaration:

```
VAR
  x: record
       i: integer;
       r: real;
       z: record
            b: boolean;
            a:array[1..5] of integer
          end
     end ;
```

U-code symbol table:

```
% PTEST3 11
V $ X S 12784 <4 D 288
! I 0 32 <5 J>
! R 32 32 <6 R>
! Z 64 224 <7 D 224
! B 0 32 <8 B>
! A 64 160 <9 A 32 <10 U 15 <5><5>>
.>
.>
#
```

Generated IR declarations[8]:

```
:↑1 F$1↑i↑1          " F$1 = i "
:↑1 F$2↑r↑1          " F$2 = r "
```

[8] The original field names have been replaced by translator generated names of the form F$n (integer n). This replacement was done because the current version of IR requires that all field names be distinct globally.

```
:↑2 F$4↑b↑1          " F$4 = b "
:↑2 F$5↑i↑5          " F$5 = a "
:↑1 F$3↑L↑rec↑1      " F$3 = z "
:↑0 X↑G↑rec↑1        " ↑rec = record "
                     "The :↑n (intcgcr n) indicates the nesting depth of a record field"
```

2.4.2 Expressions

Given below are the Fortran (infix), U-code (postfix) and IR (prefix) representations of the same expression:

Fortran statement:

```
X = a + b - c * d / (e+f)
```

U-code symbol table:

```
% $MAINB 11
V $ A    S 11224 <1 R>
V $ B    S 11260 <1>
V $ C    S 1 1296 <1>
V $ D    S 1 1332 <1>
V $ E    S 11368 <1>
V $ F    s 11404 <1>
V $ X    s 11512 <1>
#
```

U-code instructions:

```
COMM x = a+ b - c * d / (e+f)
LOD R S 11224 36
LOD R S 11260 36
ADD R
LOD R S 11296 36
LOD R S 11332 36
MPY R
LOD R S 11368 36
LOD R S 1 1404 36
ADD R
DIV R
SUB R
STR R S 11512 36
```

Generated IR code:

```
:= X - + A B / * C D + E F
```

2.4.3 Procedure Call

The major difference between a U-code procedure call and an IR procedure call is syntactic. Semantically, both representations contain the same information.

Fortran statcmcnt:

```
C P IS A DYADIC PROCEDURE, & F IS A MONADIC FUNCTION
      CALL P( 1, F(2) )
```

U-code instr-ucticns:

```
COMM   CALL P( 1, F(2) )
MST 2
LDC  J321
STR J S 1 1088 32     ; store in a temporary
LDA S 1 1088 32 1088 ; address of formal parameter
PAR A M 0 0 32
     ; start of function call F(2)
      MST 2
      LDC  J322
      STR J S 11120 32
      LDA S 1 1120 32 1120
      PAR  AM0032
      CUP  J 0 F 1 1
STR J S 1 1152 32     ; store result in a temporary
LDA S 1 1152 32 1152 ; address of formal parameter
PAR  AM03232
CUP  POP20
```

Generated IR code:

```
     :T$2↑T↑i↑1       " declare T$2 as a temporary "[9]
     :T$3↑T↑i↑1       " dcclare T$3 as a temporary "
     :T$4↑T↑i↑1       " declare T$4 as a temporary "

: =  T$2 1           " assign 1 to T$2"
:= T$3 2             " assign 2 to T$3"
: =  T$4             " assign F(2) to T$4"
     :F↑F↑i↑1
      CALL F↑P1     " call function F "
     :P$2↑M↑p↑1       " ↑M = actual parameter-"
     :=  P$2 # T$3    " # T$3 = address of T$3 "

CALL P↑P2           " call subroutine P "
     :P$3↑M↑p↑1
```

[9] These declarations will actually occur "on the fly", i.e., just before the temporary is needed, e.g., : = :T$2↑T↑i↑1 T$2 1

```
: = I'$3 # T$2    " 1st parameter = address of T$2 "
     :P$4↑M↑p↑1
: = P$4 # T$4    " 2nd paramcter = address of T$4 "
```

2.4.4 Address Arithmetic

Address arithmetic is simiiar to ordinary arithmetic, except for the LDA (Load Address) instruction and A (Address) data type in U-code, and the IR operators, *# (address of)* and @ (*dereference*). e.g.,

Fortran statement:

```
C X IS AN ARRAY DECLARED AS "INTEGER X(10)"
      X(7) = 3
```

U-code instructions:

```
COMM      X(7) = 3
LDA  S 1144 360 144
LDC  J367
IXA  J 36
DEC  Al
LDC  J363
ISTR J 0 36
```

Generated IR code[10]:

```
: =  @↑i +                  " add the offset due to the ISTR "
                          " subtract due to the DEC "
          +    X    * 7 * size↑i 1  " the IXA "
        * size↑i 1
     * 0 size↑i
  3                        " RHS = 3 "
```

2.4.5 Pascal VAR parameter declaration

Pascal Statements:

```
procedure swap (var x, y : integer);

var     t : integer;
begin
      t := x;
      x := y;
```

[10] It should be mentioned that CG always does constant folding, so that the entire esprcssion will be reduced to a constant address during code gcncration.

```
      Y: = t
end;
```

U-code symbol table:

```
% SWAP 3 2
V $ T M 3 64 <1 J>
V ↑ X M 3 0 <2 J>
V ↑ Y M 3 32 <2>
#
```

U-code instructions:

```
SWAPENT  P 2 3 2 0 0
PSTR A M 3 32 32
PSTR A M 3 0 32
COMM begin     t : = x;
LOD  AM3032
ILOD J 0 32
STR J M 3 64 32
COMM  x : =  y;
LOD A M 3 32 32
ILOD J 0 32
LOD A M 3 0 32
SWP  AJ
ISTR J 0 32
COMM  y : = t
LOD  JM36432
LOD  AM33232
SWP A J
ISTR J 0 32
COM M end;
RET
DEF M 96
END  SWAP
```

Generated IR code:

```
:SWAP ↑P2
{
    : X ↑P↑p↑1
    : Y ↑P↑p↑1
    . T ↑L↑i↑1
 : =  T @↑i +  * size↑i 0 X
 : = @↑i + X * 0 size↑i @↑i + * size↑i 0 Y
 : = @↑i + Y * 0 size↑i T
```

2.4.6 Pascal Case Statement

Pascal Statements:

```
type operator = (plusl, triple, square);

function f(op:operator; x:integer):integer;
begin
   case op of
      plusl: f : = x + 1 ;
      triple: f : = 3*x ;
      square: f: = x*x
   end
end ;
```

U-code instructions:

```
F ENT J 2 4 2 1 0
 PSTR J M 4 32 32
 PSTR L M 4 0 32
 COMM case op of
 LOD L M 4 0 32
 STR LM49632
 UJP L2
L3 LAB 0
 COMM plusl: f : = x + 1 ;
 LOD J M 4 32 32
 LDC L321
 CVT J L
 ADD J
 STR J M 4 64 32
 UJP L1
L4 LAB 0
 COMM triple: f : = 3*x ;
 LOC 1 18 0
 LDC L323
 LOD JM4 3 2 3 2 .
 CVT2 J L
 MPY J
 STR J M 4 64 32
 UJP L1
L5 LAB 0
 COMM square: f: = x*x
 LOC 1190
 LOD J M 4 32 32
 LOD J M 4 32 32
 MPY J
 STR JM46432
 UJP L1
```

```
    L6 LAB 0
     MST 2
     LDC L 32 1
     PAR L M 0 0 32
     LDC L 32 16
     PAR L M 0 32 32
     CUP P 0 $CASEERR 2 0
     UJP  L1
    L7 CLAB 3
     UJP L3
     UJP L4
     UJP L5
    L2 LAB 0
     LOD L M 4 96 32
     XJP  L L7 L6 0 2
    L1 LAB 0
     PLOD J M 4 64 32
     RET
     DEF M 128
     END  F
```

Generated IR code:

```
    :F ↑F↑i↑1 ↑P2
    {
        : OP ↑P↑i↑1
        : X ↑P↑i↑1
     : = :T$1↑T↑i↑1 T$1 OP
     got0 L$2
     L$3
     : = :T$2↑T↑i↑1 T$2 + X %↑i 1
     got0 L$1
     L$4        .
     -—  T$2* %↑i 3 X "% = convert operator"
     goto L$1
L$5
     := T$2 * X X
     got0 L$1
     L$6
   " : $CASEERR↑F↑0 CALL $CASEERR↑P2 :P$2↑M↑i↑1 : = P$2 1
     :P$3↑M↑i↑1 : = P$3 16"
     got0 L$1
     L$7
     got0 L$3
     got0 L$4
     goto L$5
     L$2
    | < T$1 0 > T$1 2 L$6
   goto + L$7 * size↑goto - T$1 0
```

```
" size↑goto is used to represent the size of a GOTO instruction.
  For example, size↑goto equals 3 in FOM because of delayed branches. "
 L$1
 : = F T$2
}
```

2.5 Translation details

Since this translation is performed in a syntax-directed fashion, the most appropriate way to specify the translation is through the context-free grammar used. The U-code grammar used is given below. Terminals are distinguished by surrounding quotes. A terminal name in capital letters denotes the corresponding U-code instruction: a lower case name indicates a terminal that stands for more than one U-code instruction. The semantic routine called performs the appropriate action depending on which instruction was actually encountered. An empty right-hand-side is denoted by ε.

```
program          ->   module

module           ->   'BGN' procedurelist singlelist 'STP'

singlelist       ->   'single' singlelist
                 ->   ε

procedurelist         procedure procedurelist


procedure        ->   'ENT' PSTRlist operationlist 'END'

PSTRlist         ->   'PSTR' PSTRlist ;
                 ->

operationlist         operation operationlist
                 ->   ε

operation        ->   expression 'popop'
                 ->   expression expression 'ISTR'
                 ->   'MST' parameterlist 'CUP'
                 ->   'single'

parameterlist    ->   parameter parameterlist


parameter             expression 'PAR'
                 ->   operation expression 'PAR'

expression       ->   expression expression 'binaryop'
                 ->   expression 'unaryop'
                 ->   'pushop'
                 ->   'MST' parameterlist 'CUF'
                 ->   expression operation
```

The last production

```
expression -> expression operation
```

is necessary because it is possible in U-code for an action to occur even if the expression stack is not empty, e.g., the STR (Store) instruction stores and pops the top entry in the expression stack, not necessarily leaving it empty.

The following productions of this grammar call semantic routines:

procedure -> 'ENT' PSTRlist operationlist 'END'

This production has two associated actions:

- shift 'ENT' - Since 'EN?" only occurs in this production, it is safe to call the procedure-entry routine whcncvcr the parser shifts over 'ENT". The routine called is *ENT−shift* in Module ACTION. *ENT- shift* essentially does two things:

    1. Calls routine *Getproctable* in Module SYMTAB to process the declarations for the current U-code procedure. *Getproctable* has to decide if any outer declarations should bc printed. It also takes care that parameters arc printed in the correct order by using the PSTR - Pseudo-STaRe - U-code instruction. After collecting all the infonnation it needs, it calls routine *Printprocdecls* in module IRWRIT to write out the IR procedure hcadcr, parameter declarations and variable declarations.

    2. Pushes thc ENT instruction on the semantic stack for use by subsequent PLOD and RET instructions that effect function/procedure returns.

- reduce - This production is reduced at the end of a U-code procedure. Routine *end−proc* in Module ACTION is called to write out the closing } for the IR proccdurc. A label is also written just before the }; all proccdurc RETURNs are simulated by a jump to this label, since IR does not have a RETURN statement. If the procedure is the main program, a STOP statement is generated. Finally, this routine pops the ENT semantic record pushed by *ENT- shift*.

operation -> expression 'popop'

reduce - Procedure *popop* in Module ACTION is called when this production is rcduccd. The production corresponds to an IR statement that must be printed. Shifting over the nonterminal *expression* guarantees that the top of thc semantic stack contains the recognized expression in some internal form (an expression tree, to be precise). The terminal 'popop' could be one of four U-code instructions:

- STR - STR corresponds to an assignment statement in IR that should be printed out in an appropriate prefix fonnat. Procedure *Printexpr* in Module ACTION does a pre-order traversal of the expression tree, to generate a prefix expression. It is possible that thc destination of the STR is a temporary, in which case a corresponding new IR variable has to bc declared.

- FJP, TJP - A conditional branch operation also pops an expression from the U-code expression stack. The boolean cxprcssion is printed in prefix form, followed by a label, to obtain a conditional branch statement in IR.

- POP - This instruction is a rarely used U-code instruction that just pops an expression from the U-code stack. It is effectively a no-op, and no corresponding IR code is generated.

- XJP - The XJP instruction is used to implement thc Pascal CASE statement by a jump table. The cxprcssion on thc stack is the selection cxyrcssion, and thc XJP instruction has thc addresses of the jump table and thc OTHERS location.

The last thing donc by *popop* is to pop the expression from the top of the semantic stack.

operation -> expression expression 'ISTR'

The ISTR instruction has 3 different syntax from STR, because it also needs the address of the destination, besides the source expression. Procedure *indirect – store* in Module ACTION is called to generate the corresponding IR statement, taking care of the appropriate destination address arithmetic. *itdirect-store* finally pops the two expression entries off the semantic stack.

operation -> 'MST' parameterlist 'CUP'

This production has both a *shift* and a *reduce* action associated with it.

- shift 'MST' - the MST instruction is merely pushed on the semantic stack, to be used when the production is reduced.

o reduce - Routine *proc – call* in Module ACTION is called to generate this procedure call. All the parameters are available on the top of the semantic stack, terminated by the MST instruction. The procedure call is written out in IR, with appropriate declaration of all "parameter temporaries". Finally, this routine must pop all the 2*n + 1 entries on top of the semantic stack, where n equals the number of parameters in a procedure call.

operation -> 'single'

This production is used for each U-code instruction that translates to an entire IR statement by itself. In most cases, a sequence of U-code instruction maps into one IR statement. Procedure *single – stmt* in Module ACTION is called to print out the IR statement in each case. There are four such U-code instructions :

1. CLAB, LAB - Print out the IR label name "L$n", where n is the integer value of the U-code label.

2. PLOD - This pseudo-load U-code instruction indicates a function return value, and a corresponding assignment to the function name, available from the ENT instruction on top of the semantic stack, is generated in the IR program.

3. RET - A procedure/function return has to be emulated in IR, by a jump to the end of the procedure/function, where a special label is defined.

4. UJP - An unconditional jump translates to a *goto* in IR.

5. SDEF - If the Static Memory Area being defined has a number > 2, it represents a Fortran COMMON, and a corresponding IR declaration must be generated.

parameter -> expression 'PAR'

The semantic action is simple in this case. The PAR instruction is pushed on top of the expression, in the semantic stack. In this way, the parameter list is built as a sequence of stack entries, ending at the top of the stack.

pa ramete r -> operation expression 'PAR'

Somctimcs an operation (most commonly an assignment to a temporary) may occur before a parameter expression is constructed. This production allows for such an operation. The operation itself will leave the stack as it found it, so the semantic action of pushing the PAR instruction, is the same in this case as the one above.

expression -> expression expression 'binaryop'

Procedure *binaryop* in Module ACTION is called when this production is reduced. It constructs a composite expression tree from the two expressions on top of the semantic stack and the given binary operator. It pops the two expressions, and pushes the constructed expression tree on the semantic stack. The following binary operators arc implemented :

ADD,AND,DIV,EQU,GEQ,GRT,IOR,IXA,LEQ,LES,MOD,MPY,NEQ,SQR,SUB,XOR.

The INST (Indirect non-destructive store) instruction is also treated as a 'binaryop' syntactically. Semantic routine *indirect-store* in Module ACTION handles this case. It generates an assignment of the expression on top of the semantic stack, to the location obtained by adding a given offset to the address expression that is second from the top. Since this is a "non-destructive" store, the source expression is pushed back onto the semantic stack, after the top two entries are popped.

expression -> expression 'unaryop'

Procedure *unaryop* in Module ACTION handles this production. Again, a composite expression tree is created that replaces the given sub-expression on top of the stack. The following unary operators are implemented :

CVT, CVT2, DEC, ILOD, INC. NEG, NOT, ODD, RND, SQR, SWP.

CVT2 is unusual, in that it operates on the expression second from top of the semantic stack. So there must be such an expression, even though the syntax does not require it. Similarly SWP exchanges the two expressions on top of the semantic stack.

As with INST in *binaryop*, the NSTR (Non-destructive store) instruction is treated like a 'unaryop' for syntactic purposes. It generates an assignment statcmcnt, just as was donc for a STR. Only, it does not pop the expression from the top of the semantic stack, since it is "non-destructive".

expression -> 'pushop'

Procedurc *pushop* in Modulc ACTION is called to create a single node expression from the given operand, and to push it on the semantic stack.

expression -> 'MST' parameterlist 'CUF'

This production is used to handle function calls. CUF is not a U-code instruction; CUP is used in U-code for both procedures and functions. However 'CUP' is replaced by the terminal 'CUF', if the CUP instruction turns out to be a call to a function. Semantic routine *func – call* in Module ACTION is called to process this function call. Unlike, a procedure call, a function call cannot bc printed immediately, as it is part of an expression tree. So the sequence of semantic stack entries from MST to the top, all have to be copied into some other data structure, since they must be popped from the semantic stack. This action is essentially what *func – call* does. Finally, it pushes an entry for the function call, as a single node expression tree, on top of the semantic stack.

3. Table-Driven Storage Binding

3.1 Introduction

In this section we discuss the issue of *operand binding* i.e., mapping variables, non-immediate constants, labels and results of expression evaluation on to storage locations in the target machine. A source language variable is expressed as an operand and it is declared with several attributes in IR [SP&E 84]. *Attributes* are used in IR to provide the *flexibility* needed to map a variety of front-ends. Furthermore, they are essential for binding variables to locations in the target architecture. They need to be converted to machine addresses before instructions are selected. Some of these attributes such as class and data type are essential for storage binding. Others such as register preference are useful for *efficient* storage binding. For example, local variables and parameters may be automatically *cached* by the target hardware. Consider the following Pascal source statement:

```
x : array[1..10] of record
                      i: integer;
                      r: real
                    end;
```

The corresponding IR is:

```
    :↑1 i ↑Local ↑integer ↑1
    :↑1 r ↑Local ↑real ↑1
:↑0 x ↑Local ↑record ↑10
```

The address of record x[10] is:

```
+ x * - 10 1 + size↑integer size↑real
```

[11]

To access the field x[10].r, the IR code is:

```
INDEX↑real + x * - 10 1 + size↑ in teger size↑real    r
```

Variables may be bound to various storage classes in the target architecture. Examples of such classes are main memory, general purpose registers, accumulators, index registers, base registers, condition code register, stack, operand queue and cache. The storage-binding phase reads in target-machine locations, architecture properties and usability restrictions. Examples of these data are memory range, number of registers, hardware stack, frame, direction of frame growth, positive and negative offsets, data types and their alignment restrictions, addressability. Sometimes, there are restrictions on binding variables to certain classes of storage. For example, variables whose addresses are needed in computation should not be allocated to ALAT

[11] size↑fieldname is also permitted.

locations in FOM. They need to be-allocated in main memory. Similarly, actual parameters of procedures cannot be allocated on the FOM stack or queue. Furthermore, because of its data type, a variable may never be present in a particular storage class. Most of these storage classes are allocatable, some are *temporarily* allocatable whereas others such as condition code register and operand queue are *not* allocatable. Usually, members of this latter category are effects of special features used to implement the target hardware, such as pipelining and operand prefetch. Such storage classes are allocated as *side effects* and may be exploited as temporaries by the instruction selection phase.

Within the category of allocatable storage classes, a number of different data types may be supported. Each data type specifics an operand length that is supported by at least one machine instruction. Due to architectural restrictions, not all data types may be supported in all storage classes. For example, not all data types may be representable in a *register* storage class on most machines. Furthermore, the choice of display/environment set up may restrict accesses and consequently, the choice of a particular storage class. For example, up-level addressing may not use displays.

In the following text we discuss storage binding as the following issues:

- Storage Description Tables that specify machine-dependent attributes such as storage class and data type. This specification describes all available storage including overlap of groups of locations that are otherwise logically distinct, alignment and addressability restrictions.

- Storage Mapping Tables that facilitate the mapping of IR attributes to machine-dcpcndent attributes.

- Expansion of variables into access primitives. Usually, the target architecture does not provide support for up-level addressing. The display/environment or static chain is a facility that is provided to access an object. An IR variable is expanded in terms of these access primitives to simplify mapping this facility to the addressing modes provided by the target architecture. However, if the target architecture does provide such a facility then expansion is not necessary[12] .

- Expansion of certain IR operators. The expansion phase also expands operators with non-mappable attributes in order to interface well with the instruction selection phase. Array address calculations that are implicit, i.e., auto-scaling of array indices as in INDEX↑s are exposed if the target architecture does not provide a facility for auto-scaling, e.g., $s \neq 1, 2$ on FOM.

[12] e.g., ULoadI mode, UpEp, VariableId in FOM

3.2 Attributes for storage binding

III variables (names, attributes) must be converted to machine *addresses* before instructions are selected. An IR *class* will map to a machine storage class, e.g., memory, stack, registers and cache. An IR *data type* will map to a machine data type, *e.g.,* byte, word, long and real. The *attribute domains* IRCLASS and IRTYPE can have the following values. These values are decided, in principle, by the I-ILL that is being compiled.

Attribute Domain	Attribute Value
IRCLASS	Global
	Static
	Local
	Formal Parameter
	Actual Parameter binding location
	Function return location
	Label
	Constant
	Compiler Generated Temporary name
	Address of variable necessary
	Import of a non-local variable or deferred size[13]
IRTYPE	Boolean
	Character
	Integer
	Real
	Pointer

An operand (datum) on the target machine will be addressable by the following triple; components of this triple are attributes:

```
<class, offset/name, access>
```

. The class attribute is the operational storage segment such as main memory, stack or register. The offset domain is the range of the storage class also called the address space of the storage segment. It is calculated as

```
start-address . . cardinality
```

of the storage class. The access domain is a machine-addressing mode, which is initially a grammar production and eventually an attribute after parsing through the addressing-mode production. The operational length of a datum is given by its data type or the address diffcrcncc between two successive elements in the corresponding storage class. The machine-dependent attributes are storage class, i.e., operational storage segment and machine data type, i.e., operand length. They are defined as follows:

CLASS: machine-specific type of physical storage location; e.g.,

---

[13] e.g., Fortran COMMON

- main memory
- general purpose registers
- stack for procedure-level or block-level allocation

These locations arc allocatablc for maximum scope and lifetimes of variables. Other locations arc *temporarily* allocatable during volatile allocation. Such volatile locations need a *temporary manager* to track contents to ensure lifetimes of variables. Examples of such classes are as follows:

- stack for expression evaluation
- temporaries in main memory
- temporaries in registers

These locations are allocated and managed by *the Temporary manager* during code generation. Some locations are not allocatable by the code generator. The hardware implicitly allocates them usually *m-the-fly*. Examples of such classes are:

- condition code bits
- cache or high speed register buffer
- Instruction-stream constant, i.e., *immediate operand.*

The following classes are examples from FOM and IBM-3701

FOM	IBM-370
Memory	Memory
Integerstack	Register
Realstack	Instructionstream
Integerqueue	
Rcalqueue	
IntegerConvertqueue	
RealConvertqueue	
IntegerALAT (ALATI)	
RealALAT (ALATR)	
Instructionstream	

These classes arc assigned to variables and remembered via an *attribute. The environment attribute* of a variable in IR is used to map the variable to a machine storage class. This class attribute must bc set up by the storage-binding phase before instruction selection. Otherwise, the instruction selection phase may fail to select an instruction and *thus, block* code generation.

DATA TYPE: the length in bits required for the variable to be an operand. Using language dependent

data type of a variable that appears as an attribute in IR, the variable is mapped to a machine specific data type. Formally, data types are groups of bits that can participate as operands to instructions; e.g.,

FOM: integer, logical. boolean, real
IBM-370: Halfword, Word, Short Float, Long Float, Decimal.

An allocatable storage class may not necessarily support all machine data types. This architectural non-orthogonality must be taken care of during storage allocation/binding and during instruction selection[14].

3.3 Storage-5escriptisn tables

The Storage description for the target architecture contains the following items:

- class and *range* of class
- *data type*
- *allocatability*
- *addressability*, i.e., address obtainability.
- *alignment* factor. Usually, on many machines, specific addresses need alignment restrictions such as, byte alignment, word alignment and long-word alignment. In DELs, such as FOM, variables arc represented by their string names and not by offsets. Conscqucntly, such alignment restrictions are not needed at the code-generator level?
- *overlapping* cells or sets of non-overlapping storage locations

The storage tables are enumerated below:

1. StartOffset:	StorageClass	→	Offset
2. BaseRegister:	S toragc Class	→	Register
3. Datatype:	IR type	→	type {Boolean,Integer,Real in FOM)
4. Bytes:	machine data type	→	number of bytes
5. Alignment:	machine data type	→	alignment number
6. StorageResidence:	IR class X machine data type → storage class		
e.g.,	Global X Integer	→	Memory

[14] *predicates* [Fischer 83] handle such restriction during instruction selection

[15] i.e., p-c-assembler level

Local X Real	→	ALATR
Formal Parameter X Reai	→	Memory

7. Allocation: Storage class X IR class → allocated result (ASM-code, Address)

e.g.,

Memory X Global	→	name: Block <size>
ALATI X Local	→	LocalI name, <initialvalue>
ALATI X Global	→	IGlobal name, <initialvalue>

3.4 Allocating space

Allocation is essentially determination of storage class and offset on the target machine or an assembler name if assembler pseudo-ops are used to allocate space. Attributes in IR and the storage-description tables are used to implement this mapping of variables to target locations. Space is allocated by mapping class and data-type attributes in IR to machine attributes: storage class and target data type. The class/environment attribute to an IR variable describes whether the variable is local, global, up-level or a parameter. Considering allocatability, e.g., architectural restrictions such as *parameters must be in main memory,* restricted register use, the variable is allocated to a particular storage class such as main memory, general purpose register or stack location. For example, FOM has the following restrictions: array element areas, common and equivalence types must be in main memory; base of the array must be in ALATI; labels and certain constants that cannot be part of the I-stream for *immediate* access.

The elementary *data type* attribute of a variable and the *how many* attribute, e.g., number of array elements, present in JR is used to determine the start address of the variable considering the minimum addressable unit/length, i.e., bit/byte/word addressability. The variable or the variable element[16] is assigned a machine-specific data type using a data-type mapping table supplied by the describer. If an architecture supports a contiguous allocatable segment of space[17] then records and arrays in IR may be allocated contiguous blocks of storage. Assembler primitives may also be used to set aside these blocks of storage. Alignment restrictions are also considered. Machine-specific offsets, base-register/location-name/location counter are assigned and entered into the code generator's symbol table or assembler primitives are used to allocate storage. Overlapping cells[18] are automatically updated as status busy.

---

[16] in the case of an array element

[17] e.g., iAPX-432 objects

[18] e.g., register pairing and sharing

determine – machine – class:

. obtain IR attribute - local, global. outer, parameter
. consider allocatability and architecture restrictions from machine-description tables,
e.g., parameters must be in main memory
. register preference - these attributes come from I-ILL front-end
. allocate selected class such as main memory, general purpose register or stack

determine- machine – datatype:

. obtain III attribute - integer, real, boolean
. use "how many" attribute in IR, e.g., array elements
. assign a machine specific data type for variable element
using data-type mapping table from machine-description tables.

allocate; obtain name or offset, class:

INPUT:
When this procedure is called, the following attributes have already
been determined: storage class, irclass, name, size/number-of-bytes,
alignment and possibly initialized value or name for this variable.
OUTPUT:
allocate either emits assembler storage primitives
or calculates an offset in the variable's storage class.

. if assembler primitives exist then
use symbolic names
emit assembler primitives for storage allocation
return;
. calculate the effect of alignment restrictions (offset + offset modulus alignment)
. determine addressability
<start address, bit/byte/word addrcssability, minimum addressable unit/length>
. determine range = start address . . max- allocatablc address of class
. determine address-difference
overlapping cells will be automatically updated as status busy,
e.g., register pairing and sharing
. assign offsets, base register/name/location – counter and enter into symbol table.

3.5 Operand access

All attribute information regarding a variable is kept in the code-generator's symbol table. Thus, attributes of variables in IR need be declared only once. This symbol table supports block structure. For up-level addressing, access is specified in terms of *indexing* and *indirection* primitives. The run-time environment is set up using a display, a static-chain mechanism or an environment vector. Correspondingly, a frame pointer, a display pointer or an environment pointer is selected from among the machine locations. Using this environment and access mechanism, any reference to an IR variable is *expanded* using *indirection* and *indexing* primitives. IR names and attributes arc expanded to machine addresses using multiple levels of indirection and indexing. Thus, opportunities for optimization in address arithmetic arc exposed prior to

instruction selection. Examples of such optimizations arc the use of hardware indexing to subsume (i.e., compile-back) additions and scaled multiplications, c.f., VAX-11, Motorola-MC68000 architectures. Indexing can be defined in terms of indirection. The scaling factor is an attribute to these operators.

INDEX↑s vara varb = *INDIR ECT*↑s + vara * varb size↑s

thus, when size↑s = 1; *INDEX*↑ s vara varb = *INDIR ECT*↑s + vara varb

and when varb = 0, indirection can be visualized as a special case of indexing;

INDEX↑ s vara 0 = *IND IR ECT*↑s vara

Consider the following example using a static chain environment in the FOM architecture:

: varA ↑up-level↑2levels[19] ↑integer ↑1unit "a static-chain example"

To achieve up-level addressing in FOM, an ALATI entry is used to store the environment pointer (stack frame location) of the parent procedure that must have been previously invoked. The variable itself is expanded into a chain of pointers:

INDEX varA INDIRECT INDIRECT framepointer↑ALATI

This expanded string may be mapped to any one of the following possibilities:

- a single variable with its auto-indirect bit ON
- use of the variable as an address and load its contents
- load and store with the EP (environment pointer) of the referenced procedure[20]

If the scaling factor *s* is not directly mappable to the target architecture's scaling factor then

INDEX↑s A I is expanded to INDIRECT↑s + A * I size↑s

---

[19] JR requires that the number of levels be explicitly specified. This information can be deduced from its symbol table but then dynamic binding cannot be supported.

[20] as suggested in [Brantley 82]

Import of variables

If a variable is not declared in the current block, it must be *imported.* Examples of such cases are global variables that are declared outside the current block, and forward references of variables declared outside the current block.[21] In such cases, the variable must be declared with an attribute ↑I before its first use in the current block. This process must be repeated in every procedure the variable is used in a sitnilar fashion. In niost cases,

": var ↑Import" is equivalent to ": var ↑Local ↑pointer ↑ 1 initialized[22] with the variable's address".

3.6 Address Obtainability

IR provides two operators for address arithmetic. These operators are # (address of) and size↑t, where *t* is any type including record names and component field names. Array indexing is performed by arithmetic using these operators. An array is guaranteed to be allocated as a contiguous block, so this kind of relative address arithmetic is always valid.

Sometimes, the necessity for obtaining the address of a variable occurs *on-the-fly* during instruction-selection. For example, such necessities arise when passing arguments to functions by address in Fortran and when passing VAR parameters in Pascal. Some storage locations on the target machine may not be address-obtainable. Examples of these classes are general-purpose registers on most machines and the ALAT in FOM.[23] There does not exist an addressing mode[24] or a load-address instruction[25] that can provide the address of the variable resident in this storage class. When an address has to be copied using the *Bind* instruction in FOM, the source ALAT location must be initialized with the corresponding address. This copy is impossible if the address being passed is that of an ALAT object. Consequently, any object whose address is required cannot be allocated on the ALAT. It must be allocated in memory[26] and then an ALAT entry should be initialized with its address. In order to ensure this allocation, the IR provides an attribute ↑*A* that must occur with a variable whose address would be needed. before the first use of the variable in the current scope.

---

[21] e.g., Fortran Common. Forward references in Pascal are not permissible on other machines.

[22] The code generator will emit a LocalI var, var$Ba for every such procedure.

[23] ALA?' is not address-obtainable because of hardware implementation issues.

[24] e.g., immediate address on the PDP-11

[25] e.g., LA on the IBM-370 and MOVA(bwlfqd) on the VAX-11

[26] e.g., using the block declaration

Even if the address of a variable is available in the storage location in which the variable is resident, a problem may arise if this storage location cannot participate as an operand to the operating instruction.[27] For example, consider scalar VAR parameters of type real. FOM parameters are implemented with the BindR instruction that copies ALATR locations. If the caller's ALATR location contains an address, then the single word address is copied. But there is no way for the callee to extract a single word from a double word ALATR location. For example, a *CnvR* op-code is used to convert a real variable to an integer variable.

```
CnvR a.., X, Fix%
```

does not yield the least-significant-word of the double-word ALATR entry, X. Instead, it converts the nutnerical value of X to a fixed-point reprcscntation. The problem is that the address is available in the caller's ALATR, but it needs to be supplied in a *BindI* instruction that requires its operands to be in ALATI. Therefore, an ALATI entry should be created, initialized with the address, and used for the BindI.[28] Thus, any time the address of a real variable is needed, a fresh local temporary[29] is allocated and initialized with the global address of the real variable. This temporary, being in ALATI, is directly used in BindI. For example:

```
LocalI temp, %X
LocalR X, %X
Bind1 temp, 3
```

3.7 An Example: FOM Storage description

- Main memory, allocatable
- NO registers
- NO stack for block level or procedure level allocation
- Stack for expression evaluation; volatile storage class
- NO condition code bits
- High Speed Buffer called ALAT; allocatable with restrictions, i.e., excluding arrays, common variables and in general whenever the address of a variable is required.,
- Large constant (small constants as immediate operands); allocatable

---

[27] This restriction is an architectural restriction on the programming model.

[28] Ideally, in FOM, the VAK parameter should be implemented by an AutoIR. But then LoadR cannot be used to access it: instead it should be accessed directly. Furthermore, the code generator does not have the information to recognize a scalar, real, VAR parameter, i.e., the IR declaration is ↑Parameter↑pointer↑1, which could mean a pointer to an integer or a real.

[29] This yields cstrancous tcmporarics in ALATI. Since all locals are "cached" by FOM, there arc no performance problems due to these extra tcmporarics.

- Queue; volatile

Storage Class	*Allocatable bit/byte/word* addressable	*Addressability* startaddress..finaladdress	Allocatable range
Main memory	yes	word(16 bits)	
Stack	no	word(16 bits)	top, top-l
Queue	no	word(16 bits)	front, front-15
Alat	yes	word(16 bits)	
Constant	yes	Immediate	-64 .. 63

There is NO ccl! ovcrlapping, i.e., different names do not exist for the same physical location.

Data type	width	*Alignmen t factor* modulus alignment
Integer	16	mod 16'
Real	32	mod 32
Boolean	16	mod 1
Logical	16	mod 1

Storage Mapping Tables for FOM

Storage Class	Offset	Base Register
Memory	0	None
IntegerStack	0	None
RealStack	0	None
IntegerQueue	0	None
RealQueue	0	None
IntegerConvertQueue	0	None
RealConvertQueue	0	None
ALATI	2	None
ALATR	0	None
InstructionStream	0	None

IRtype	FOM Da tatype
Boolean	Boolean
Character	Integer
Integer	Integer
Real	Real
Pointer	Integer

FOM Datatype	Bytes	Alignment
Boolean	1	1
Tntcgcr	1	1
Rcal	2	1

IRclass	Boolean	Integer	Real
Global	Memory	Memory	Memory
Static	Memory	Memory	Memory
Local	ALATI	ALATI	ALATR
Formal Parameter	ALATI	ALATI	ALATR
Actual Parameter	ALATI	ALATI	ALATR
Function Return	ALATI	ALATI	ALATR
Label	ALATI	ALATI	ALATR
Constant	ALATI	ALATI	ALATR
Temporary	ALATI	ALATI	ALATR
Address Obtainable	Memory	Memory	Memory

Allocation

(Memory, Global)	→	"\n%s:\tBlock %o"
(Memory, Static)	→	"\n%s:\tBlock %o"
(Memory, Address)	→	"\n%s:\tBlock %o"
(ALATI, Local)	→	"\tLocalI %s, %s\n"
(ALATI, Formal P)	3	"\tLocalI %s\n"
(ALATI, Function)	→	"\tAutoII %s\n"
(ALATI, Label)	3	"\tDclLabel %s\n"
(ALATI, Constant)	→	"\tLocalI %s, %o\n"
(ALATI, Temporary)	→	"\tLocalI %s, %s\n"
(ALATI, Address)	→	"\tIGlobal %s, %s\n"
(ALATR, Local)	→	"\tLocalR %s\n"
(ALATR, Formal P)	→	"\tLocalR %s\n"
(ALATR, Function)	→	"\tAutoIR %s\n"
(ALATR, Constant)	→	"\tLocalR %s, %o, %o\n"
(ALATR, Temporary)	→	"\tLocalR %s\n"
(ALATR, Address)	→	"\tRGlobal %s, %s\n"

4. FOM Machine Grammar and the Instruction Selection Phase

4.1 Data types and grammar non-terminals

FOM supports the following *data types*:

- Integers, 32 bits wide
- Logical, 32 bits wide; the most significant bit (bit 0) contains the logical. The remaining 31 bits of the word are ignored.
- Boolean, 32 bits wide; this word is interpreted as a vector of 32 binary bits.
- Floating point, 64 bits wide.

Grammar non-terminals are used to represent each of the above data types. These non-terminals are italicized below:

Data Type	Grammar Non-Terminal
In tegcrs	*Infeger*
Logical	*Integer*
Boolean	*Ii2teger*
Floating point	*Real*

The following section enumerates FOM instructions, restrictions on their operand USC and access, and corresponding machine grammar productions. The *most general* production is listed last. This production represents the most general form of use for the corresponding instruction. This production may or may not appear with a blocking predicate depending on the instruction-set architectural restrictions on the programming model. If a production appears with a blocking prcdicatc, then the production is applicable if the blocking predicate evaluates to *true*. To prevent the code generator from blocking, productions must also be supplied with the same syntactic form but without the blocking predicate. These productions will be selected if the architectural restrictions specified by the blocking predicate are not satisfied for a syntactically valid IR input.

Later, other productions that represent optimal use of the instruction under various contexts arc described. These optimizations are *data flow* dependent and are represented by identical productions with different disambiguating predicates. The *synthetic attribute* ↑r contains information pertaining to the result of a machine operation. Machine-dependent details such as the variable location, such as stack, queue, register, memory, cache, and the specific address within that location form part of the attribute ↑r.

4.2 Grammar productions

To handle Affix Grammars. i.e., semantic attributes and predicates, the parser driver was modified to accommodate *context-sensitive* aspects [Fischer 83] and. thus, provide *context-sensitive* pattern matching that is needed for instruction selection. Thus, deriving a code generator.is very similar to deriving an attributed context-fret parser. Generating code is very similar to doing attributed parsing [Toplas 843. Productions use terminals and non-terminals with attributes. Prodicatcs control production application. Action symbols compute attributes and emit code. As an example, consider the FOM addressing-mode production:

Addressf LDQ R → INDEX↑scale Integer↑a Integer↑b *scale=2*
EMIT ↓'LoadR'↓a↓ b

where Integer and Address are grammar non-terminals with attributes 'a' and 'b' specifying locations that participate in the indexing operation. Emit is an action symbol that emits a LoadR instruction to synthesize the attribute LDQR, i.e., result in the queue, for this datum on FOM. The predicate *scale=2* specifies FOM-architecture restrictions on synthesizing this code. If the predicate evaluates to *false*, recognition of this production is blocked. Consequently, a subsequent production is matched that satisfies architectural restrictions. Examples of similar restrictions on other architectures are IBM-370 restrictions on displacement and base-register use, iAPX-86 restrictions due to segmentation and index register use, and in gcncral, if the attribute 'b' happens to be a memory location instead of a register on most machines.

The use of *semantic attributes* facilitates a *type-sensitive* machine grammar. This feature is an important factor in reducing the grammar size. Furthermore, inclusion of type information at the grammar level permits implicit type coercions to be driven by the grammar. This facility is especially important to support mixed-mode arithmetic with a number of arithmetic types in the HLL. Other semantic information such as intermediate results, i.e., temporaries in expression evaluation, immediate constants or memory references are also handled as semantic attributes to grammar symbols. To perform individual operand accesses, Addressing-mode productions discover feasible uses of effective address generation mechanisms of the target machine. Consequently, an operand-access format is selected.[30]

Address↑a →	Datum↑a	*a. class* = *Instructionstream* a.format = "%d" i.e., value
Addressf a →	Datum↑a	*a.class* = *ALAT* a.format = "%s" i.e., name
Address↑a →	Datum↑a	*a.class* = *Stack* a. format = " " i.e., blank space
Addressf a →	Datumf a	*a.class* = *Queue* a.format = "Load%Q" i.e., load queue

[30] The format then becomes a semantic attribute to the Lhs non-terminal Address.

The non-terminal Address reduces to another non-terminal that signifies a machine data-type in FOM:

Integer↑a	→	Address↑a	*a. type = integer*
Real↑a	→	Address↑a	*a. type = real*

Predicates are a very useful aid in making parsing decisions. This aid allows us to use grammars that otherwise could not be parsed. Such predicates are often termed **disambiguating** predicates because they resolve parsing decisions that otherwise would be ambiguous. For example, the following productions represent situations in which some form of division may be generated. These productions match the IR; they generate FOM instructions via EMITs.

Real↑r	→	/ Real↑r Real↑1	
Real↑r	→	/ Real↑a Real↑b *Constant*↓*a* ∧ *Constant*↓*b*	KFOLD↑r = a/b
Real↑r	→	/ Realf a Real↑r ¬ *Busy*↓*r*	EMIT↓'DivR'↓a↓r↑r
Real↑a	→	/ Real↑a Real↑r ¬ *Busy*↓*a*	EMIT↓'DivR'↓a↓r↑a
Real↑r	→	/ Real↑a Real ↑b *Stack*↓*'TOP'*↓*b* ∧ *Stack*↓ *"TOP-I'*↓*a* Temp↓'real'↑r	EMIT↓'RDiv'↓b↓a↑r
Real↑r	→	/ Realf a Real↑b Temp↓'real'↑1	EMIT↓'DivR'↓a↓b↑r

The predicates *Constant* and *NotBusy* perform a two-fold function. First, they serve as a guide to when a production is applicable and second, they serve to control parsing, i.e., resolve reduce-reduce conflicts that would otherwise occur if this grammar were predicate-free. The first two productions recognize the special case of a division by the constant one and division of a constant by another. The third and fourth productions investigate if the locations containing either of the operands is not busy and thus can be used to store the result. A divide instruction is generated. The next production checks if the operands are in reverse order on the FOM expression stack, thus requiring a 'reverse-divide' instruction. The last production is the final "match all" production that will obtain a temporary to store the result and then emit a divide.

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 productions so that their predicates are evaluated first. The thrust of this addition is to facilitate incremental development of an optimizing code generator. The inclusion of such productions contributes to shift-reduce conflicts in the machine grammar. To resolve such conflicts, *predicates* are used. These predicates may be *contextual* predicates or *look-ahead* predicates. *Contextual predicates* examine the current context of operation; all relevant context being available on the attribute stack of the code-generator-generator. *Look-aheadpredicates* examine the look-ahead symbol, if any, already provided by the code-generator parser. This examination is necessary to prevent *blocking* of the code generator for a valid IR input.

For simple *instruction selection* productions, *a* final *match-all* production, with no predicate restriction, suffices to prevent blocking. On the contrary. for *optimization productions*, blocking resolution must be done by adding productions with *converse* predicates for each *non-lookahcadpredicate* that evaluates to *false* and follows *a lookahead predicate* that evaluates to *true*. In FOM such cases do not arise.

Finally, *we use Cost predicates* to select among different implementations of the same instruction set that have different speeds for instructions, operands, addressing modes and cache effects. Such instruction implementations are usually represented as identical productions that differ only in *Cost predicate* values. However, cases arise when *Cost predicates* are needed to resolve conflicts on the basis of instruction timings. In such cases, they appear in non-identical productions. For example, in FOM, a

```
NorB  a, b, c
```

is preferable to a

```
OrB a, b, c; NorB 0, c, c
```

Thus, it seems likely that the *NorB* reduction should always be preferred to an *OrB* reduction for an IR string

```
NOT OR a b
```

However, the code generator may choose to implement

```
NOT OR a b label
```

as either

```
OrB a, b, c;   IfLF c, label
```

or

```
NorB a, b, c;   IfLT c, label
```

The code generator will USC cost predicates to select the lesser of

time[OrB]+*time*[If LF]

and

time[NorB]+*time*[If LT]

The complete FOM grammar is listed on an instruction-by-instruction basis in the Appendix.

4.3 Grammar Issues

Operands may need to be relocated intentionally by the code generator to storage locations other than the one they normally reside. Such relocations may or may not be associated with a corresponding change in the machine data-type. Usually, deficiencies in instruction-set orthogonality and the inherent design of two-address instructions that perform destructive operations[31] lead to operand relocations with no associated change in their data type. Examples of such occurrences are:

- lack of memory-to-memory operations on the iAPX-86 and the 28000 microprocessor architectures.

[31] i.e., one of the operands is replaced by the result of the operation

- impossibility of memory-to-memory arithmetic on the IBM-370 and the PDP-11 computers.
- two-address arithmetic operations that destroy the contents of one of the operands; consequently, one of the operands must be moved to a temporary location before the operation is performed.

Such operand relocations are handled by *blocking predicates* and *disambiguating predicates.*

On non-tagged target architectures, the data-type encoding is part of the op-code specifier. Mixed-mode operations are, therefore, implemented by converting all operands to the same machine data-type before performing the operation. Such conversions may be specified explicitly as type coercions by the compiler front-end or performed implicitly by the code generator. To implement forced operand relocations associated with data type conversions, *LoopCheck* predicates are used. For example, to convert an integer datum to a real datum on FOh4, the following production is used:

Real↑Cnv%Q → Integerfa *CheckConvert*↓*a*↓ *'Integer'*↓ *'Real'*
EMIT↓'CnvI'↓a↓Float%

The predicate *CheckConvert* is used to check if conversion from integer to real format is really needed in the current context of operation. Context is determined by interrogating the attribute stack of the code generator. The desired left sibling will usually be at a constant offset from the top of the stack for all items in the configuration set. The absence of the *transfer production* may force the code generator to block while processing a semantically correct IR input. The absence of the predicate alone will result in a shift-reduce conflict of the predicate-less production with other *instruction selection* productions such as

Integer↑*r* → + *Integer*↑*r Integer*↑*a EMIT*↓*'AddI'*↓*a*↓*r*↓*r.*

The inclusion of this *transfer production* may cause loops in the code-generator automaton when data type conversions are performed more than once for the same variable. For example, an integer variable may be converted to a real variable and then subsequently converted back to an integer variable by a sequence of reductions 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 con figuration of the code generator.

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, *If-Logical-False* on FOM. 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 such fancy instructions as grammar productions with a longer right hand side. For example, the IfLF instruction in FOM can be specified as:

```
Instruction  → ¬ Boolean↑b Label↑n
                    EMIT↓'IfLF'↓b↓n
                    EMIT↓'Nop'
                    EMIT↓'Nop'
```

Consider the IR string ¬ *b n* (*if not b then goto label n*) and a shift-reduce conflict of the above *optimization production* with the following *instruction selection* production that describes a *Nor Boolean* instruction on FOM:

```
Boolean↑b  → ¬ Boolean↑b ¬ Busy↓ b
                    EMIT↓'NorB'↓0↓b↑b
```

For purposes of this discussion, let us assume that *Busy↓ b* evaluates to false. There exist two derivation possibilities.

- The *NorB* production is selected followed by an *IfLT* production:

```
Booleanf b  → ¬ Boolean↑b ¬ Busy↓ b
                    EMIT↓'NorB'↓0↓b↑b
Instruction → Booleanfb Label↑n
                    EMIT↓'IfLT'↓b↓n
                    EMIT↓'Nop'
                    EMIT↓'Nop'
```

- The *IfLF* production is selected:

```
Instruction  → ¬ Booleanfb Label↑n
                    EMIT↓'IfLF'↓b↓n
                    EMIT↓'Nop'
                    EMIT↓'Nop'
```

The *IfLF* is a better choice than *NorB, IfLT* and even more so if *Busy↓ b* evaluates to *true* in which case a temporary allocation could be saved too.

In view of these two derivations, the code-generator-generator reports a shift-reduce conflict between the *NorB* and *IfLF* productions. If the code generator always shifts, i.e., selects an *IfLF*, then it may block for a correct IR string such as ":= *A* ¬ *b*". Similarly, if the code generator always reduces, i.e., selects a *NorB*, then poor quality (but correct) code will be generated.

To resolve this issue correctly, an *IfLF* should be selected if the *immediate context* of operation is a conditional. Otherwise a *NorB* should be selected. Because the IR is prefix, the *immediate context* is the left context of the current configuration set that is always available on the stack of the code-generator parser. Predicates can be used to examine the current context at calculated positions on the stack where a needed left

sibling is stored. Furthcrmorc, in practice, because of the nature of instruction-selection productions, the desired left context will usually be at a constant offset from the top of the stack for all items in the configuration set. Thus, *contextual predicates* arc added to the conflicting productions as follows:

```
Instruction   → ¬ Boolean↑b context = conditional Label↑ n
                    EMIT↓'IfLF'↓b↓n
                    EMIT↓'Nop'
                    EMIT↓'Nop'
Boolean↑b     → ¬ Boolean↑b ¬ Busy↓ b ∧ context ≠ conditional
                    EMIT↓'NorB'↓0↓b↑b
```

In this case, there can be some minor optimization in the code-generator speed. The *context* need be examined only if the *look-ahead* at the conflicting point is a *label. A look-ahead predicate LAshift* could be inserted before the *contextual predicate.* The predicate *LAshift* evaluates to *true* if the look-ahead symbol already provided by the parser happens to be one among its arguments. In this example, the look-ahead symbol must be a *label* so that the shift *may possibly* be taken in place of the reduction. The actual shift will take place pending *true* evaluation of the *contextual predicate.* Thus, the productions can also be written as follows:

```
Instruction   → ¬ Boolean↑ b LAshift↓ label ∧ context = conditional Label↑ n
                    EMIT↓'IfLF'↓b↓n
                    EMIT↓'Nop'
                    EMIT↓'Nop'
Boolean↑b     → ¬ Boolean↑b ¬ Busy↓ b ∧ context ≠ conditional
                    EMIT↓'NorB'↓0↓b↑b
```

In summary, the instruction-selection phase performs the following code-generation functions:

- It selects machine addressing modes.
- It Selects target machine instructions. Sometimes, instructions are subsumed within addressing modes or they are constant folded within address arithmetic.
- It performs temporary allocation and data type conversions where needed. To handle non-orthogonality of the target-architecture's instruction-set, addressing mode conversions and other restrictions imposed by the target machine are handled by predicates to grammar productions -- the control is automated through the parsing scheme.

4.4 Implementing Predicates and Code Generation Algorithm

Not all grammar forms are easily parseable. Therefore, in practice, only certain classes of grammars, most commonly LL(1) and LALR(l) [Aho 73], are used. Due to the prefix nature of IR and since real computers often have numerous instructions that can be used to effect the same result, *top down* parsing is not well-suited to target code generation. Bottom-up parsing is preferred. For a rationale, the reader may refer to

[Ganapathi 80]. In this paper, we concentrate on adding predicates to LALR(1) parsers, using YACC[Johnson 75] as a foundation. For details on implementing predicates in *top down* and other *bottom up* parser generators, the reader is referred to [Fischer 83].

In many ways, a predicate symbol, i.e., a symbol that represents the invocation of a predicate, can be viewed as a variety of *terminal symbols*. The predicate symbol represents a marker verifying the semantic constraints of the symbols on the right-hand-side of a grammar production. If the semantic constraints are not satisfied, the marker is absent, and the production is blocked. This approach is easily implemented. We examine the parse table entries corresponding to a given parser configuration. If a predicate symbol can be read in a given parser configuration then the corresponding predicate is evaluated. All predicates are assumed to be side-effect free. If the predicate evaluates to *true*, a token corresponding to the predicate symbol is inserted into the input. Look-ahead symbols must be saved prior to insertion of the predicate symbol. The insertion allows the parser to consume it and production recognition proceeds. If the predicate evaluates to *false*, the predicate symbol is not inserted, and the production is blocked, possibly causing a syntax error to be recognized. In effect, special markers are added to the user's input as a side-effect of predicate evaluation, providing extra information to the parser.

In LR-type parsers, predicates that appear anywhere except at the extreme right can be implemented as either terminal symbols, or as new non-terminals that derive only ε. Implementation of predicates as non-terminals is attractive in that predicate evaluation can be triggered by the usual production-recognition mechanism. Further, these non-terminals can cause no look-ahead problems because they generate only e.

Another issue is the use of predicates as *look aheads*. *A look ahead* is a terminal symbol that may not be part of the production being matched, but rather part of the context just beyond it. Predicates should only be *visible* and evaluated when they are part of a production currently being matched. Their use as look-aheads must be severely limited. In particular, all predicate symbols that appear as look aheads for reduction actions must be removed or avoided.

The code-generation algorithm mirrors the standard LR(k) parsing loop with added code to implement predicates.

```
PROGRAM Code-generator;

State : = 0;
Action : = Shift;

    SWITCH (Action) OF

        CASE Shift:
            Push(State);
            IF Look-ahead-Token exists      THEN
                                        Symbol : = Look -ahead-Token
                                    ELSE
                                        Symbol : = Lexicalanalyser();
            Action : = Nextaction(State, Symbol);
            State : = Nextstate(State. Symbol);
            IF Buffer ≠ Empty THEN
                BEGIN
                    Look-ahead-Token : = Buffer;
                    Buffer : = Next symbol in Buffer
                END
        END;

        CASE Reduce:
            Pop(RHS-of-production);
            State : = Top-of-stack;
            Action : = Nextaction(State, LHS-of-production);
            State : = Nextstate(State, LHS-of-production);
        END;

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

        CASE Error:
            (* halt, rejecting *)
        END

    END

END.
```

4.5 Dynamic Conflict Resolution

Consider the following productions:

P1: A → C
P2: B → C
P3: D → E A B

P4: D → E B A

A *reduce-reduce* conflict is caused by P1 and P2. To *disambiguate* this conflict, we add a non-terminal Nonterm, a disambiguating routine disambiguate and tvkcns Tokena and 'l'okcnb as follows:

P1: A → C Nonterm Tokcna
P2: B → C Nontcrm Tokenb
P5: Nonterm → ε
disambiguatc(*Attributestack*,Tokena,Tokenb)

The decision to select P1 or P2 is made by *disambiguate* when a reduction by P5 *triggers* the disambiguating predicate. This predicate looks at the context, the attribute-stack or uses any conditions programmed by the user and inserts either Tokena or Tokcnb in the parser's input stream as an indication of its choice.

The parser, if it uses look-ahead, may have already read a look-ahead token. In this case, the *disambiguating token* must be inserted before any such look-ahead token and the latter must be saved in a token *Buffer*. For a *k look-ahead* parser, *Buffer* has to be a queue of depth *k* so that all *k* parser look-aheads can be saved in the buffer queue before *disambiguate* inserts any token in the parser's input stream. The following code illustrates this process.

```
PROCEDURE Disambiguatc (Token1, Token2,...., TokenN)
    BEGIN
        IF Prcdicatel THEN insert(Token1)
        ELSE    IF Predicate2 THEN insert(Token2)
                ELSE .....
    END;

PROCEDURE insert(Token)
    BEGIN
        IF Look-ahead-Token THEN Buffer : = Look-ahead-Token;
        Look-ahead-Token : = Token
    END
```

4.6 Output Formatting

After instructions arc sclccted, a few files[32] are used for output buffering. Tables are used to specify these files and also to provide output formatting.

Variable-Prefix: Storage Class X IR Class → Prefix String

(Memory, Global) → "$Ba"
(Memory, Static) → "$Ba"

[32] These files may bc in-core files or disk files or a combination of the two. This implementation choice depends on the storage limits of the system on which the code generator is resident.

(Memory, Function)	→	"$F"
(M emory, Constant)	→	"$C"
(Memory, Temporary)	→	"$T"
(Memory, Address)		"%"
(ALATI, Global)	→	"$Ba"
(ALATI, Static)	→	"$Ba"
(ALATI, Function)	→	"$F"
(ALATI, Constant)	→	"$C"
(A LATI, Temporary)	→	"$T"
(ALATI, Address)	→	"%"
(A LATR, Global)	→	"$Ba"
(ALATR, Static)	→	"$Ba"
(ALATR, Function)	→	"$F"
(ALATR, Constant)	→	"$C"
(ALATR, Temporary)	→	"$T"
(ALATR, Address)	→	"%"

File: Storage Class → file number (0 . . 7)

Addressing-mode Prefix: Storage Class → symbol {'a','k','q','s'}

Storage Class	File	Addressing-mode Prefix
Memory	0	'a'
IntegerStack	5	's'
RealStack	5	's'
IntegerQueue	5	'q'
RealQueue	5	'q'
IntegerConvertQueue	5	'q'
RealConvertQueue	5	'q'
ALATI	2	'a'
ALATR	3	'a'
InstructionStream	5	'k'

Prolog: File → String

0	→	"\n"
1	→	"\nBegProc "
2	→	"\nALATI\n\tLocalI Ret$IP\n\tLocalI Ret$EP\n"
3	→	"\nALATR\n"
4	→	"\nEntryProcs IP$"
5	→	"\n\t"
6	→	"\nAlatHeader EP$"
7	→	"\nEndProc "

Run-time Start-Up

```
runstart        → "\n.insert fom2.fai\n.insert fommac.fai";
runend          → "\nstopFOM \nEND";
extinscrt       → "\n.insert fio.fai\n.insert frt.fai\n";
```

4.7 An Example of Code Generation by Attributed Parsing

Consider integer arrays A, B and C and the following Fortran assignment statement:

A(1) = B(J) + C(K)

The Attributed Polish Prefix Linear Intermediate Representation is:

: = INDEX↑integer A I + INDEX↑integer B J INDEX↑integer C K

The next phase sets the environment, assigns storage classes, such as ALATI, ALATR, memory, stack, and binds variables to storage locations in FOM. Information regarding storage classes, machine data-types, alignment restrictions etc. are provided as separate tables. After, *the storage binding and expansion phase,* every variable possesses a machine data type[33] with attributes specifying the storage class and the machine-specific location for that variabie. Thus, in the current example, the IR is expanded into:

```
: = INDEX↑ 1 Integer↑'ALATI',a Integer↑'ALATI',i
+ INDEX↑1 Integer↑'ALATI',b Integer↑'ALATI',j INDEX↑1 Integer↑'ALATI',c Integer↑'ALATI',k
```

The next phase performs instruction selection. We now illustrate attributed parsing and the code generated for the FOM architecture. The code that is generated upon production recognition, i.e., a reduction, is enclosed within *{}*. The following lines trace the parsing process.

```
[1]  : = INDEX↑ 1 Integer↑'ALATI',a Integer↑'ALATI',i
     + INDEX↑1 Integer↑'ALATI',b Integer↑'ALATI',j INDEX↑1 Integer↑'ALATI',c Integer↑'ALATI',k

[2]  : = INDEX↑ 1 Integer↑'ALATI',a Integer↑'ALATI',i
     + Address↑LdQI {LoadI aa., b, j} INDEX↑1 Integer↑'ALATI',c Integer↑'ALATI',k

[3]  : = INDEX↑1 Integer↑'ALATI',a Integer↑'ALATI',i
     + Integer↑ LdQI INDEX↑ 1 Integer↑'ALATI',c Integer↑'ALATI',k

[4]  : = INDEX↑ 1 Integer↑'ALATI',a Integer↑'ALATI',i
     + Integer↑LdQI Address↑LdQI {LoadI aa., c, k}

[5]  : = INDEX↑ 1 Intcgerf 'ALATI',a Integer↑'ALATI',i
     + Integer↑LdQI Integer↑LdQI

[6]  : = INDEX↑ ] Integer↑'ALATI',a Integer↑ 'ALATI',i
     + Integer↑LdQI Integer↑LdQI Temp↓'integer'↑StoTemp
```

[33] *Integer* in the current example

[7] : = INDEX↑ 1 Integer↑'ALATI',a Integer↑'ALATI',i
Integer↑StoTemp *{AddI qqa, Load%Q, Load%Q, StoTemp}*

[8] : = INDEX↑1 Integer↑'ALATI',a Integer↑'ALATI',i Integer↑StoTemp *Lastref↓ 'ALATI'↓StoTemp*

[9] Instruction *{StoI aa., a. i}*

Thus, the FOM code generated for the Fortran assignment statement is:

```
Load1 aa., b, j
Load1 aa., c, k
AddI qqa, Load%Q, Load%Q, StoTemp
StoI aa., a, i
```

5. Transient Observations

5.1 Unsupported features - possible further extensions

Both U-code and IR arc considered to be language-independent intermediate forms. Conscqucntly, a translator from U-code to IR should not need any knowledge of the HLL used to obtain the U-code. But this observation is not so in practice. Occasionally, the semantics of an HLL feature are defined with some machine-dependent assumptions. Such features are very troublesome in a compilation scheme with an intcrmcdiatc form, because the machine dependence has to be somehow implemented in the machine-independent intermediate language. This problem does not occur in monolithic compilers where the translation is performed directly to machine language.

The Ucodc-to-IR translator is intended to work with both Fortran and Pascal. U-code features that have not been implemented arc described below, along with remarks about what their implementations would involve:

- Runtime support - The problem with runtimc libraries is that some of them arc necessarily machine-dependent. This dependency includes procedures for I/O, dynamic memory allocation, runtimc errors, etc. Machine-independent libraries could bc supported by having an IR library of the procedures, e.g., Math routines, but machinedependent routines cannot be written in IR. In such a case, it is unavoidable **that** there be separate copies of these routines for different machines. However, it is simple for IR to support separate compilation, so that. programs that use such routines can be compiled through IR, even though the routines are not written in IR. This solution is used in the U-code system as well.

- Sets - Sets were not implemented because they arc not supported in the current implementation of IR. It would be a simple extension to include them by introducing vector boolean operators. Of course, this operation can always be expressed by using a FOR loop, but that would lead to an inefficient implementation on some machines.

- Non-local GOTOs - This feature has not been implemented, mainly because it is not supported by FOM. Translating a non-local GOT0 from U-code to IR is straightforward. The problem occurs in the back-end that has to implement it. Non-local GOTOs can be easily incorporated in both *static chain* and *display register* schemes. However, neither of these traditional schemes can be efficiently implemented in FOM.

- Procedure parameters - They have not been implemented for the same reason as non-local GOTOs.

- String variables - String variables is a Pascal* [Hennessy 79] feature that we have not supported.

- Import/export - Again, import/export of variables is a Pascal" feature that we have not implemented.

- Data initialization - The current implementation of IR does not have a compile-time data initialization construct. Of course, a data initialization can always be mimicked by an assignment statement, but that would happen at run-time rather than at compile-time. This feature can be implemented if data initialization was included in the I R implementation.

- Global register allocation - The current implementation of TR has no provision for specifying registers.[34] Therefore, the translator does not support U-code instructions that operate on registers. This operation can only be done if a limited machine-dependency is permitted in the IR implementation, as is done in U-code.

- Block move/compare - The only way to express block move/compare operations in IR is by generating the equivalent FOR loop, which is inefficient. The translator could support these operations once they are available in IR.[35]

5.2 Code Generator Statistics

The FOM code generator is resident on the VAX-11/780[36] running Unix[37] and it occupies 100 K bytes, mostly data space. It produces about 100 lines of FOM assembler code per second. The grammar incorporates 184 symbols and 267 grammar productions. The Code-Generator-Generator (CGG) takes about two minutes to produce the FOM code generator. The compilers have been rctargeted to the IBM-370. The IBM-370 code generator occupies 120 K bytes. The grammar incorporates 200 symbols and 430 grammar productions. The CGG takes about four minutes to produce the EM-370 code generator.

5.3 FOM Simulation Measurements

This section describes the results obtained by simulating 6 benchmark programs (2 Fortran & 4 Pascal) on the ISPS simulator [Barbacci 77], driven by an ISP description of FOM. The measurement taken was the number of instructions cxecutcd. Both the unoptimized and optimized FOM code were simulated for each benchmark. As described carlicr, the optimizations were performed at the U-code level by UOPT.[38] *Register allocation* was the only UOPT optimization disabled for FOM, as FOM has no registers.

Due to the slow performance of the TSPS simulator (approximately 15 FOM instructions per CPU-second

---

[34] This feature is not important for FOM.

[35] e.g., by attributing the : = operator with size of the block to bc moved

[36] VAX is a trademark of Digital Equipment Corporation.

[37] Unix is a trademark of Bell Laboratories.

[38] The performance improvement obtained is about 50% for the DEC-10, MC-68000 and the VAX-11. Register Allocation is often responsible for 30% of the optimization improvement.

on a DEC-20), we were forced to drastically reduce the size of the benchmark programs. This naturally leads to a smaller improvement due to optimization, because of the relative increase in time spent on initialization and I/O. Our results are therefore pessimistic about the potential speedup obtainable by machine-independent code optimization.

INVERT

INVERT is a matrix inversion program, written in Fortran. As a benchmark, it is designed to invert a 10x10 real matrix for 200 iterations. We used only 1 iteration on a 3x3 matrix, and obtained these results:

Unoptimized code:	2344 instructions
Optimized code:	2171 instructions
Percentage improvement:	7%

QUICKSORT

QUICKSORT is also a Fortran benchmark, which uses an explicit stack instead of recursion. It was originally written to sort an array of 300 integers for 100 iterations. We used 1 iteration on an array of 5 integers:

Unoptimized code:	927 instructions
Optimized code:	841 instructions
Percentage improvement:	9%

BUBBLESORT

BUBBLESORT is a Pascal program, written to sort an array of 70 integers for 100 iterations. We used 1 iteration on an array of 10 integers:

Unoptimized code:	2136 instructions
Optimized code:	18 52 instructions
Percentage improvement:	13%

PRIME

PRIME is a prime number generator written in Pascal. It uses the "Sieve of Erastosthenes" method, and was written to generate all prime numbers smaller than 16384, for 50 iterations. We generated prime numbers smaller than 104, for 1 iteration:

Unoptimized code:	1881 instructions
Optimized code:	1908 instructions
Percentage improvement:	-1%

The "pessimization" of -1% is due to an unsuccessful attempt at *strength reduction* by the optimizer. The multiplication in an array subscript turned out to be a multiplication by 1 that generated no code in the unoptimized case. The optimizer attempted to reduce this multiplication to an addition by introducing a temporary. In this case, it was a pessimization because of the extra code generated to increment the temporary.

INTMM

INTMM is an integer matrix multiplication benchmark written in Pascal. It was designed to multiply two 40x40 integer matrices (‘just once). We used two 3x3 matrices instead:

Unoptimized code:	1756 instructions
Optimized code:	1565 instructions
Percentage improvement:	11%

MM

MM is just like INTMM, except that it uses floating-point numbers instead. Again we used two 3x3 matrices instead of the original 40x40 size:

Unoptimized code:	2332 instructions
Optimized code:	1805 instructions
Percentage improvement:	*23%*

In comparison, the performance improvements obtained arc about 50% for the DEC-10, MC-68000 and the VAX-11. We are in the process of obtaining similar data with the IBM-370 CG.[39]

5.4 Code Tradeoffs

A number of tradeoffs can be made in the generated FOM code. Important ones are itemized below:

- The use of Autoindircction facilitates compact code space but is a drain on time. An alternate solution would be to generate immediate stores to memory and loading from memory using Store and *Loud* instructions.

- Some FOM op-codes are non-orthogonal to data types; i.e., they do not operate on all data types provided by FOM. For example, comparison operators operate on integer data types only. The code generator (CG) will automatically convert such operations on other data types to the valid operation by converting the operands to the valid data type.[40] This solution may lead to inaccuracies, especially when floating-point data is converted to integer data. Thus, an alternate solution would be to provide grammar productions for operations on other data types with sequences of target instructions that implement the operation.[41]

- Rcversc-stack op-codes do not exist for certain non-commutative operators.[42] Consequently, either the expression stack should not be used for such operations or grammar productions must .

---

[39] First, WC need to obtain an assembler and a simulator for the IBM-370.

[40] In the machine grammar, such operators must appear with an attribute that specifies the valid data type.

[41] Subtraction is a good tcchniqx to perform real comparison.

[42] RSub and RDiv exist but reverse comparison op-codes are absent in FOM.

be supplied that reflect reverse-stack operations with instructions using opposite opcodes[43] or opposite results?

5.5 Tracking

To utilize machine locations efficiently and also to produce correct code in the presence of architectural restrictions, CG tracks the contents of certain storage classes.[45] On many architectures, these locations tend to be general-purpose registers, top few locations on the stack, condition codes and other beneficial locations in the processor state whose contents are valid between instructions. Apart from ensuring architectural restrictions on the operand USC, tracking also yields optimizations such as auto-increment, auto-decrement on registers, avoiding redundant loads and stores, recognizing potential aliases in memory and register. In FOM, the following locations are tracked:

Storage Class	locations
Stacks	top, one below the top[46]
Queues	front, next in queue"[47]
ALATI	last reference to ALATI
ALATR	last reference to ALATR

The *tracked* information is retained as semantic attributes, to be used during attributed parsing.

5.6 Context-Specific Temporary Allocation

To improve target-code quality, *context* is propagated through semantic attributes. The main advantage of the *context* attribute is that the target-machine describer can *incrementally* program the extent to which context-dependent optimization may be pcrformcd. In particular, it is a convenient method of ensuring safety in the presence of optimizations.[48] Context specific temporary allocation is performed as itemized below:

---

[43] e.g., LtI for GeqI, GtI for LeqI and vice-versa

[44] e.g., IfLF for IfLT and vice-versa

[45] Sometimes, depending on the necessity, CG can also track temporaries. The main thrust of this step is to potentially overload unavoidable temporaries that are not common-sub-expressions. Thus, the temporary store is re-used and in consequence, the total number of temporaries is often minimized -- see [Wulf 82].

[46] Could track more locations if more than 2 stack locations are provided and indexing off the stack-top is allowed.

[47] Tracking two locations suffices for correctness but tracking more queue locations could yield optimal code.

[48] Two examples of such interferences arc: (a) Code scheduling intcrfcrcs with the use of the stack for temporaries, e.g., scheduling of LoadI sk.,..... may not be safe: and, (b) Incomplete grammar specification interfcres with the use of the stack without revcrsc-stack op-codes.

- To reduce ALAT bank conflicts, the expression stack may be used instead of ALAT locations. In the context of addition, subtraction, multiplication and division, the use of the stack, for temporaries, yields optimal code. Depending on allocation, such uses may or may not involve reverse-stack operations as depicted in the following examples:

```
DivR sas, , Alattempl,                  DivR aaa, Alattempl, Alattemp2, Alattempl
RSub ssa, , , Alattemp2                 SubR sas. , Alattempl,
AddR kaa, 0, Alattemp2, Result          AddR ksa, 0, , Result
```

- If the machine grammar does not contain productions that reflect reverse-stack operations for comparison operators, then in the context of comparison operators, the expression stack is not used for storage of temporary results.

- In a *store* context, the value to be stored must be the last ALA?' reference. Consequently, ALAT temporaries are preferred to stack temporaries. Use of the stack will introduce a move to an ALA?' location before the final store.[49]

- In *index*, *indirection* and *conversion* contexts, data gets placed in one of the FOM Queues. These instructions are likely to be re-scheduled.[50] Stack temporaries need not be *live* across the code-motion window, but ALAT temporaries are always *live*.[51]

5.7 Code Scheduling

Unlike code generation for other architectures,[52] in FOM, CG provides some minimal code scheduling. The FOM architecture provides queueing disciplines[53] that forces CG to provide code reorganization. Generating correct, leave aside optimal, code for queues can be quite tricky, e.g., for the expression $a + b * c$, a naive code generator may generate:

```
Load a                  Enqueue a in queue Q
Load b                  Enqueue b in queue Q
Load c                  Enqueue c in queue Q
Multiply Q, Q, R        Multiply first 2 Q elements - also dequeue
                        them - and store the result in location R
Add Q, R, R             Add head of Q to R
```

Because of the FIFO property of a queue, this code evaluates $a * b + c$ instead of $a + b * c$. Usually, such wrong schedules occur within the domain of an assignment statement only.

---

[49] Add ksa, 0, , ALATlocation

[50] Code Scheduling is discussed in the next section.

[51] There are only 2 stack locations per data type and they get re-used often: however, there are 128 ALAT entries that get block-loaded on procedure entry. Thus, no attempt is made to w-use/overload ALA?' temporaries across instructions.

[52] IBM-370, VAX-11, iAPX-86, L-8000, MC-68000, PDP-11.

[53] Load-Store and Convert Queues.

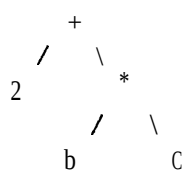
Scheduling can be performed in a- separate post-pass optimization phase or in tcgrated during code generation itself. One advantage of the in tegrated scheduling approach is that it still fits into a *single pass* code generation scheme and the other advantage is the availability of *aliasing* information in attributes. Usually, the critical problem in code scheduling is to find, and handle, aliasing information. Incomplete knowledge about potential aliasing conflicts restricts reordering more than other factors. Another strong motivation for incorporating scheduling within CG was the use of the existing *tracking* mechanism. Specifically, in FOM, CG tracks the last ALAT reference, stack top and next-below entries and queue front and next-in-queue cntrics. In CG, integrated scheduling is performed as a two step process by:

- tracking the front and next II entries in qucucs, where n is maximum arity of op-codes in the target machine,[54] and
- performing code motion (specifically code delaying) depending on the context of the current operation.

In the realm of attributed parsing, consider the following example: When parsing

$$\ldots\ldots Op_1\ Operand\uparrow Queue_1\ Op_2 Operand\uparrow Queue_2 \ldots\ldots$$

the operation on $Queue_1$ should be delayed until after operations on Queue operands for op_2. This delay ensures that op_2 operates on the correct data. In effect, such a schedule simulates a bottom-up version of breadth-first traversal of the expression tree,[55] e.g., the tree for $a + b * c$ is:



The bottom-up breadth-first traversal scans levels from left to right, starting at the bottom. The nodes visited and the code generated are as follows:

Visit b	Load b
Visit c	Load c
Visit 2	Load a
Visit *	Multiply Q, Q, R
Visit +	Add Q, R, R

To obtain maximum concurrency, this strategy can be extended to maximize the distance between an Enqueue and the corresponding Dequeuc instructions, exploiting a longer queue. Furthermore, to take

---

[54] For FOM, n = 2; thus, it suffices to track the front and next entries.

[55] In a tree-walk evaluator, the bottom-up traversal can bc awkward to perform, and so a top-down approach can be used that generates code in the reverse order. It would have to scan each level from right to left.

advantage of FOM's parallelism,[56] non-queue instructions such as *NxtProc* should bc *hoisted* as far as possible.[57] Other optimizarions of similar flavor would be Nop removals.[58] Nops do not cost performance but they cost space.

5.8 Conclusion

Given current compiler technology, DEL architectures seem promising. Traditionally, the optimization of code for DEL compilers had been largely unexplored. The reverse-synthesis process is therefore an interesting approach to measure optimizations on DEL architectures. Another advantage of this approach is rctargctability and rehostability. Although FOM is an architecture originally designed for Fortran, it can also be used for Pascal. The reverse-synthesis process may seem to destroy the go21 of reducing the semantic gap between the source program 2nd the target program. Given the up-down-up translation process, it is hard to *guarantee* that the JR to target-architecture translation is not worse than direct source to target translation. However, the effects of various optimizations 2nd empiric21 results do reveal that DEL architectures and this *compile-back* technology can co-operate to produce efficient translation 2nd high performance.

This experience has suggested 2 number of techniques by which CG can be potentially useful to a computer architect. These items are enumerated below and shall be addressed in detail in future research.

- CG can identify areas in which the grammar is incompletely specified. This knowledge can reveal extra instructions needed in the target machine's instruction-set architecture.
- The machine grammar can provide a measure for static code-size that has 2 second order effect on performance.
- In order to measure the architecture, the *compiler effect* is fixed by incorporating productions in the most general form only. Predicates are first used to specify architectural restrictions only 2nd not for optimization. With this grammar, 2 code generator is created 2nd then the architecture is "benchmarked".
- Optimization productions arc then incorporated incrementally 2nd the architecture is "benchmarked" in the presence of optimizations.

---

[56] executing instructions in parallel with loading of ALAT

[57] NxtProc should not be hoisted beyond a basic block or another NxtProc.

[58] For example, safely *hoisting* the comparison and branch instructions two instructions before their current position within the domain of straight-line control flow.

Acknowledgements

We thank Bill Brantley and Fred Chow for their technical help during the course of our research and Mike Flynn for his comments.

6. References

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

- Barbacci 77
  M.R. Barbacci, G.E. Barnes, R.G. Cattcll and D.P. Siewiorek, "The ISPS Computer Description Language". Technical Report, Department of Computer Science, Carnegie-Mellon University, 1977.

- Brantlcy 82
  W.C. Brantlcy and J. Weiss, "FOM: A Fortran Optimized Machine - A High Performance, High Level Language Machine", IBM Research Report KC 9640 (#40815) 3/3/82. Also, "A Fortran Optimized Machine A High Performance, High Level Language Machine", Proceedings of the International Workshop on High-Level Language Computer Architecture, Nov. 30 - Dec. 3, Fort Lauderdale, Florida, 1982.

- Brantley 83
  W.C. Brantley and J. Weiss, "Organization and Architecture Trade-offs in FOM", IBM Research Report RC 9700 (#42748) 11/11/82. Also, Proceedings of the International Workshop on Computer Systems Organization, March 29 - 31, New Orleans, Louisiana, 1983.

- Fischer 83
  C.N. Fischer, M. Ganapathi and R.J. LeBlanc, "A Simple and Practical Implementation of Predicates in Context-Free Parsers", Technical Report #493, Computer Scicnccs Department, University of Wisconsin - Madison, 1983.

- Flynn 80
  M.J. Flynn, "Directions and Issues in Architecture and Language", IEEE Computer, October 1980.

- Flynn 83
  M.J. Flynn and L.W. Hoevel, "Execution Architecture: The DELtran experiment", IEEETC, Vol. C-32 No. 2, February 1983.

- Ganapathi 80
  M. Ganapathi, "Rctargctablc Code Generation and Optimization using Attribute Grammars", PhD dissertation, Technical Report #406, University of Wisconsin - Madison, 1980.

- Hennessy 79
  J.L. Hennessy, "Pascal*", Technical Report, Computer Systems Laboratory, Stanford University.

- LALR
  J.L. Henncssy, "The Stanford Pascal Parser Generator", Computer Systems Laboratory, Stanford University.

- Johnson 75

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

• Koster 71
C.H.A. Koster, "Affix Grammars", in J.E.L. Peck (editor), ALGOL 68 Implementation, North Holland. 1971.

• Notices 83a
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, 1983. pp. 27 - 34.

• Notices 83b
F.C. Chow and M. Ganapathi, "Intermediate Languages in Compiler Construction -- A Bibliography", ACM SIGPLAN Notices Vol. 18 No. 11, 1983, pp. 21 - 23.

• PL.8 82
M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler", SIGPLAN 82 Symposium on Compiler Construction, June 1982.

• Popl 82
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.

• SP&E 84
M. Ganapathi and C.N. Fischer, "Attributed Linear Intermediate Representations for Retargetable Code Generators", Software - Practice and Experience, to *appear,* April 1984.

• Surveys 82
M. Ganapathi, C.N. Fischer and J.L. Hennessy, "Retargetable Compiler Code Generation", ACM Computing Surveys, Vol. 14, No. 4, December 1982.

• Surveys 83
M. Ganapathi, C.N. Fischer and J.L. Hennessy, "Retargetable Code Generators", ACM Computing Surveys, Surveyors Forum, Vol. 15, No. 3, September 1983.

• Toplas 84
M. Ganapathi and C.N. Fischer, "Instruction Selection by Attributed Parsing", Technical Report #256, Computer Systems Laboratory, Stanford University, February 1984; *to appear in ACM Transactions on Programming Languages and Systems.*

• U-code 82
U-code: an intermediate representation for Pascal* and Fortran, Peter Nye, Computer Systems Laboratory, Stanford University, November 1982.

• Ugen 82
Ugen: a retargetable code generator for U-code, Peter Nye, Computer Systems Laboratory, Stanford University, November 1982.

- u-opt 83
  U-opt: a machine-independent global optimizer, Fred Chow, Phd Thesis, Computer Systems Laboratory. Stanford University, December 1983.

- Uwtr 81
  M. Ganapathi, C.N. Fischer, S.J. Scalpone and K.C. Thompson, "Linear Intermediate Representation for Portable Code Generation", Technical Report # 435, University of Wisconsin - Madison, 1981.

- Wakefield 53
  S. Wakefield, PhD Thesis, Computer Systems Laboratory, Stanford University, 1983.

- Wulf 82
  W. Wulf and K.V. Nori, "Delayed Binding in PQCC", Technical Report, Computer Sciences Department, Carnegie-Mellon University, October 1982.

7. Appendix A: FOM Grammar

Operations on *Integers* yielding an *Integer* result:

- AddI: Add Integer

```
Integer↑ r →   + Integer↑0 Integerfr
Integer↑r  →   + Integer↑r Integer↑0
Integer↑r  →   + Integer↑a Integer↑b Constant↓a ∧ Constant↓b
                                        KFOLD↑r = a+b
Integer↑ r →   + Integer↑a Integer↑r ¬ Busy↓r
                                        EMIT↓'AddI'↓a↓r↑r
Integer↑ a →   + Integer↑a Integer↑r ¬ Busy↓a
                                        EMIT↓'AddI'↓a↓r↑a
Integer↑ r →   + Integer↑a Integer↑b Temp↓'integer'↑r
                                        EMIT↓'AddI'↓a↓b↑r
```

- SubI: Subtract Integer

```
Integer↑ r →   - Integer↑r Integer↑0
Integerf r →   - Integerf a Integerf b Constant↓a ∧ Constant↓b
                                        KFOLD↑r = a-b
Integer↑r  →   - Integer↑a Integer↑r ¬ Busy↓r
                                        EMIT↓'SubI'↓a↓r↑r
Integer↑a  →   - Integer↑a Integer↑r ¬ Busy↓a
                                        EMIT↓'SubI'↓a↓r↑a
Integerf r →   - Integer↑a Integerf b Temp↓'integer'↑r
                                        EMIT↓'SubI'↓a↓b↑r
```

- ISub: Reverse Subtract Integer

```
Integer↑ r →   - Integer↑a Integer↑b
               Stack↓'TOP'↓b ∧ Stack↓'TOP-1'↓a Temp↓'integer'↑r
                                        EMIT↓'ISub'↓b↓a↑r
Integer↑ r →   - Integerf a Integer↑b
               Queue↓'FRONT'↓b Queue↓'FRONT-1'↓a Temp↓'integer'↑r
                                        EMIT↓'ISub'↓b↓a↑r
```

- MpyI: Multiply Integer

```
Integerfr →    * Integer↑1 Integer↑r
Integer↑ r →   * Integer↑r Integerf 1
Integer↑ r →   * Integerfa Integer↑b Constant↓a ∧ Constant↓b
                                        KFOLD↑r = a*b
Integerf r →   * Integer↑a Integerfr ¬ Busy↓r
                                        EMIT↓'MpyI'↓a↓r↑r
Integerf a →   * Integer↑a Integerfr ¬ Busy↓a
                                        EMIT↓'MpyI'↓a↓r↑a
Integer↑r  →   * Integer↑a Integer↑b Temp↓'integer'↑r
```

EMIT↓'MpyI'↓a↓b↑r

- DivI: Divide Integer

```
Integer↑ r →   / Integer↑r Integer-T 1
Integer↑ r →   / Integer↑a Integer↑ b Constant↓a A Constant↓ b
                                        KFOLD↑r = a/b
Intcgcrfr →    / Integer↑a Integer↑r ¬ Busy↓r
                                        EMIT↓'DivI'↓a↓r↑r
Integer↑ a →   / Integer↑a Integer↑r ¬ Busy↓a
                                        EMIT↓'DivI'↓a↓r↑a
Integer↑r  →   / Integcrfa Integer↑b Temp↓'integer'↑r
                                        EMIT↓'DivI'↓a↓b↑r
```

o IDiv: Reverse Divide Integer

```
Integer↑ r →   / Integer↑a Integer↑ b
                 Stack↓'TOP'↓ b ∧ Stack↓ TOP-I '↓a Temp↓'integer'↑r
                                        EMIT↓'IDiv'↓b↓a↑r
Integer↑r  →   / Integer↑a Integer↑b
                 Queue↓'FRONT'↓b Queue↓ FRONT-I '↓a Temp↓'integer'↑r
                                        EMIT↓'IDiv'↓b↓a↑r
```

- LoadI: Load Integer

```
Address↑LdQI → INDEX↑1 Integer↑a Integer↑b
                                        EMIT↓'LoadI'↓a↓b
```

The attribute '1' of the operator INDEX indicates that the index 'b' must be scaled by a multiplication factor of 1 (i.e., each element of the array occupies 1 word on the FOM machine). If this attribute is a constant other than '1' or '2', the storage-allocator will expand the index operation into a combination of addition and scaled multiplication, i.e.,

INDEX↑s Integer↑a Integer↑b where s ≠ 1 or 2 is expanded into
INDIRECT↑s + Integer↑a * Integer↑b Integer↑s

Consequently, a 'MpyI' *may be* generated followed by an 'AddI'. If's' is a constant, * *Integer↑ b Integer↑ s* will be *constant folded* into *Integer↑ b*s* and the expression is reduced to:

INDIRECT↑s + Integer↑a Integer↑b*s

- StoI: Store Integer

```
Instruction  →  : = INDEX↑1 Integer↑a Integer↑b Integer↑c Lastref↓'ALATI'↓c
                                        EMIT↓'StoI'↓a↓b
```

The blocking predicate *Lastref* evaluates to *true* if its second attribute was the last referenced variable in the storage location denoted by the first attribute. If this predicate evaluates to *false*, one of the following productions is matched:

```
Instruction  →  : = INDEX↑1 Integer↑a Integer↑b Integer↑c Storage↓'ALATI'↓c
                                        EMIT↓'AddI'↓0↓c↓c
```

EMIT↓'StoI'↓a↓b

The blocking predicate *Storage* evaluates to *true* if its second attribute is a member of the storage class denoted by its first attribute.

Instruction → : = INDEX↑1 Integer↑a Integer↑b Integerfc Temp↓'ALATI'↓'integer'↑r
EMIT↓'AddI'↓0↓c↑r
EMIT↓'StoI'↓a↓b

Grammar factoring can be used to avoid replication of code. In particular, predicates may be shared by different productions to minimize code. If other productions also happen to use similar predicates, then the total number of gratnmar productions may bc reduced at the expense of grammar *readability* and *retargetability.*

Operations on *Integers* yielding a *Boolean* result[59] :

- LtI: Less than - Integer

Integerfr → < Integer↑a Integer↑b *Constant*↓*a* ∧ *Constant*↓*b*
KFOLD↑r = a<b
Integer↑ r → < Integerf a IntegerTr ¬ *Busy*↓*r*
EMIT↓'LtI'↓a↓r↑r
Integer↑ a → < Integer↑a IntegerTr ¬ *Busy*↓*a*
EMIT↓'LtI'↓a↓r↑a
Integer↑ r → < Integer↑a Integer↑b Temp↓'integer'↑r
EMIT↓'LtI'↓a↓b↑r

- LeI: Less than or equal to - Integer

IntegerTr → < = Integer↑a Integerfb *Constant*↓*a* ∧ *Constant*↓*b*
KFOLD↑r = a<=b
Integer↑ r → < = Integer↑a IntegerTr ¬ *Busy*↓*r*
EMIT↓'LeI'↓a↓r↑r
Integerf a → < = Integer↑a Integer↑ r ¬ *Busy*↓*a*
EMIT↓'LeI'↓a↓r↑a
Integer↑ r → <= Integer↑a Integerfb Temp↓'integer'↑r
EMIT↓'LeI'↓a↓b↑r

- NeqI: Not equal - Integer

Integer↑ r → ≠ In tegcr↑ a In tegcr↑ b *Constant*↓*a* ∧ *Constant*↓*b*
KFOLD↑r = a#b
Integer↑ r → ≠ Intcgerf a IntegerTr ¬ *Busy*↓*r*
EMIT↓'NeqI'↓a↓r↑r
Integer↑a → ≠ Intcgerf a Integcrf r ¬ *Busy*↓*a*
EMIT↓'NeqI'↓a↓r↑a
Integer↑ r → ≠ Integer↑a Integer↑b Temp↓'integer'↑r

---

[59] TOM does not provide reverse-stack operations for comparison operators. However, grammar productions can use predicates to check reverse-stack operations. On production selection, either the opposite op-code is emitted or the result of comparison is reversed.

EMIT↓'NeqI'↓a↓b↑r

- GtI: Greater than - Integer

```
Integer↑r   →   > Integer↑a Integer↑b Constant↓a ∧ Constant↓b
                                          KFOLD↑r = a>b
Integer? r  →   > Integer↑a Integer↑r ¬ Busy↓r
                                          EMIT↓'GtI'↓a↓r↑r
Integer↑ a  →   > Integer↑a Integer↑r ¬ Busy↓a
                                          EMIT↓'GtI'↓a↓r↑a
Integer↑r   →   > Integer↑a Integer↑b Temp↓'integer'↑r
                                          EMIT↓'GtI'↓a↓b↑r
```

- GeI: Greater than or Equal to - Integer

```
Integer↑ r  →   >= Integer↑a Integerfb Constant↓a ∧ Constant↓b
                                          KFOLD↑r = a>=b
Integer↑ r  →   > = Integer↑ a Integer↑r ¬ Busy↓r
                                          EMIT↓'GeI'↓a↓r↑r
Integer↑a   →   > = Integer↑a Integer↑r ¬ Busy↓a
                                          EMIT↓'GeI'↓a↓r↑a
Integer↑ r  →   > = Integer↑a Integer↑b Temp↓'integer'↑r
                                          EMIT↓'GeI'↓a↓b↑r
```

- EqI: Equal to - Integer

```
Integer↑r   →   = Integer↑a Integer↑b Constant↓a ∧ Constant↓ b
                                          KFOLD↑r = a=b
Integer↑ r  →   = Integer↑a Integer↑r ¬ Busy↓r
                                          EMIT↓'EqI'↓a↓r↑r
Integer↑a   →   = Integer↑a Integer↑r ¬ Busy↓a
                                          EMIT↓'EqI'↓a↓r↑a
Integer↑r   →   = Integer↑a Integer↑ b Temp↓'integer'↑r
                                          EMIT↓'EqI'↓a↓b↑r
```

Operations on *Floating points* yielding a *Floating point* result:

- AddR: Add Real

```
Real↑r      →   + Real↑0 Real↑r
Real↑r      →   + Rcalfr Real↑0
Real↑r      →   + Real↑a Real↑b Constant↓a ∧ Constant↓ b
                                          KFOLD↑r = a+ b
Real↑r      →   + Real↑a Real↑r ¬ Busy↓ r
                                          EMIT↓'AddR'↓a↓r↑r
Rcal↑a      →   + Real↑ a Rcalfr ¬ Busy↓a
                                          EMIT↓'AddR'↓a↓r↑a
Real↑r      →   + Real↑a Real↑b Temp↓'real'↑r
                                          EMIT↓'AddR'↓a↓b↑r
```

• SubR: Subtract Real

```
Real↑r    →  - Real↑r Real↑0
Realfr    →  - Real↑a Real↑b Constant↓a A Constant↓b
                                          KFOLD↑r = a-b
Real↑r    →  - Real↑a Real↑r ¬ Busy↓r
                                          EMIT↓'SubR'↓a↓r↑r
Realfa    →  - Real↑a Real↑r ¬ Busy↓a
                                          EMIT↓'SubR'↓a↓r↑a
Real↑r    →  - Real↑a Real↑b Temp↓'real'↑r
                                          EMIT↓'SubR'↓a↓b↑r
```

• RSub: Reverse Subtract Real

```
Realfr    →  - Real↑a Real↑b
             Stack↓'TOP'↓b A Stack↓'TOP-1'↓a Temp↓'real'↑r
                                          EMIT↓'RSub'↓b↓a↑r
Real↑r    →  - Real↑a Real↑b
             Queue↓'FRONT'↓b Queue↓'FRONT-2'↓a Temp↓'real'↑r
                                          EMIT↓'RSub'↓b↓a↑r
```

• MpyR: Multiply Real

```
Realfr    →  * Real↑1 Real↑r
Real↑r    →  * Real↑r Real↑1
Real↑r    →  * Real↑a Real↑b Constant↓a ∧ Constant↓b
                                          KFOLD↑r = a*b
Realfr    →  * Real↑a Real↑r ¬ Busy↓r
                                          EMIT↓'MpyR'↓a↓r↑r
Real↑a    →  * Real↑a Realfr ¬ Busy↓a
                                          EMIT↓'MpyR'↓a↓r↑a
Rcalfr    →  * Real↑a Realfb Temp↓'real'↑r
                                          EMIT↓'MpyR'↓a↓b↑r
```

• DivR: Divide Real

```
Real↑r    →  / Real↑r Real↑1
Real↑r    →  / Real↑a Real↑b Constant↓a A Constant↓b
                                          KFOLD↑r = a/b
Real↑r    →  / Rcalfa Real↑r ¬ Busy↓r
                                          EMIT↓'DivR'↓a↓r↑r
Real↑a    →  / Real↑a Realfr ¬ Busy↓a
                                          EMIT↓'DivR'↓a↓r↑a
Real↑r    →  / Real↑a Realfb Temp↓'real'↑r
                                          EMIT↓'DivR'↓a↓b↑r
```

• RDiv: Reverse Divide Real

```
Real↑r    →  / Real↑a Real↑b
```

Stack↓'TOP'↓b ∧ Stack↓'TOP-1'↓a Temp↓'real'↑r
EMIT↓'RDiv'↓b↓a↑r

Real↑r → / Real↑a Real↑b
Queue↓'FRONT'↓b Queue↓'FRONT-1'↓a Temp↓'real'↑r
EMIT↓'RDiv'↓b↓a↑r

- LoadR: Load Real

Address↑LdQR → INDEXf2 Integer↑a Integerfb
EMIT↓'LoadR'↓a↓b

The attribute '2' of the operator INDEX indicates that the index 'b' must be scaled by a multiplication factor of 2 (i.e., each element of the array occupies 2 words on the FOM machine). If this attribute is a constant other than '1' or '2', the storage-allocator will expand the index operation into a combination of addition and scaled multiplication, i.e.,

INDEX↑s Integer↑a Integer↑b where s ≠ 1 or 2 is expanded into
INDIRECT↑s + Integer↑a * Integer↑b Integer↑s

Consequently, a 'MpyI' *may be* generated followed by an 'AddI'. If's' is a constant, ** Integer↑b In teger↑* s will be *constant folded* into *Integer↑b*s* and the expression is reduced to:

INDIRECT↑s + Integer↑a Integer↑b*s

- StoR: Store Real

Instruction → : = INDEX↑2 Integer↑a Integer↑b Real↑c *Lastref↓'ALATR'↓c*
EMIT↓'StoR'↓a↓b

The blocking predicate *Lastref* evaluates to *true* if its second attribute was the last referenced variable in the storage location denoted by the first attribute. If this predicate evaluates to *false*, one of the following productions is matched:

Instruction → : = INDEX↑2 Integer↑a Intcgerfb Real↑c *Storage↓'ALATR'↓c*
EMIT↓'AddR'↓0↓c↓c
EMIT↓'StoR'↓a↓b

The blocking predicate *Storage* evaluates to *true* if its second attribute is a member of the storage class denoted by its first attribute.

Instruction → : = INDEX↑2 Integer↑a Integer↑b Realfc Temp↓'ALATR'↓'real'↑r
EMIT↓'AddR'↓0↓c↑r
EMIT↓'StoR'↓a↓b

Grammar factoring can be used to avoid replication of code. In particular, predicates may be shared by different productions to minimize code. If other productions also happen to use similar predicates, then the total number of grammar productions may be reduced at the expense of grammar *readability* and *retargetability.*

FOM does not provide comparison opcodes for *floating point* data types. Therefore, code must be explicitly

provided with productions for floating point comparison@' , or, the code generator will *automatically* transform floating point comparisons to integer comparisons.

Operations on *Booleans and Logicals* yielding a *Boolean* or *Logical* result:

- AndB: And Boolean

```
Integer↑0   →  & Integer↑0 Integer↑r
Integer↑0   →  & Integer↑r Integer↑0
Integer↑r   →  & Integer↑1 Integer↑r
Integer↑r   →  & Integer↑r Integer↑1
Integer↑r   →  & Integer↑a Integer↑b Constant↓a ∧ Constant↓b
                                        KFOLD↑r = a&b
Integer↑r   →  & Integer↑a Integer↑ r ¬ Busy↓r
                                        EMIT↓'AndB'↓a↓r↑r
Integer-f a →  & IntegerTa Integcrfr ¬ Busy↓a
                                        EMIT↓'AndB'↓a↓r↑a
Integer↑r   →  & IntegerTa Integer↑b Temp↓'integer'↑r
                                        EMIT↓'AndB'↓a↓b↑r
```

- OrB: Or Boolean

```
Integer↑r   →  | In teger↑0 Intcgerfr
Integer↑  r →  | Integer↑r Integer↑0
Integer↑  1 →  | Integer↑ 1 Integerfr
Integer↑  1 →  | Integer↑r Integer-f 1
Integer↑  r →  | Integer↑ a Integer? b Constant↓a ∧ Constant↓b
                                        KFOLD↑r = a|b
Integer↑  r →  | IntegerTa Integer↑r ¬ Busy↓r
                                        EMIT↓'OrB'↓a↓r↑r
Integer↑  a →  | Integer↑a Integer↑r ¬ Busy↓a
                                        EMIT↓'OrB'↓a↓r↑a
Integer↑r   →  | Integerfa Integer↑ b Temp↓'integer'↑r
                                        EMIT↓'OrB'↓a↓b↑r
```

- NorB: Nor Boolean

```
Integer↑  0 →  ¬ | Integerf 1 Integer↑1
Integer↑0   →  ¬ | Tntcgcrfr Integer↑1
Integer↑r   →  ¬ | Integer↑a Integer↑b Constant↓a ∧ Constant↓b
                                        KFOLD↑r = ¬ a|b
Integer↑r   →  ¬ | IntegerTa Integerfr ¬ Busy↓ r
                                        EMIT↓'NorB'↓a↓r↑r
Integerf  a →  ¬ | IntegerTa Integer↑r ¬ Busy↓a
                                        EMIT↓'NorB'↓a↓r↑a
Integer↑  r →  ¬ | Integer↑a Integer↑b Temp↓'integer'↑r
```

---

[60] for example, SubR aa$ a, b; CnvR s.., , Signum%; IfLT qa., Cnv%Q, Label

EMIT↓'NorB'↓a↓b↑r

Operations on *Integers* yielding a *floating point* result:

- CnvI: Convert Integer

  Real↑CnvQR → Integerfa

  EMIT↓'CnvI'↓a↓'Float%'

Operations on *Floating points* yielding an *integer* result:

- CnvR: Convert Real

  Integer↑CnvQI → Real↑a

  EMIT↓'CnvR'↓a↓'Fix%'

Local Control-Flow Operations

The two instructions **that** follow a branch instruction (called the *subject instructions*), are always executed. Therefore, in the moss general form, these instructions are emitted with two 'Nops' following them. By use of *instruction buffering*, Nop removal optimization may be accomplished wherever feasible.

- Goto: Go to

  Instruction → Goto Label↑n

  EMIT↓'Goto'↓n
  EMIT↓'Nop'
  EMIT↓'Nop'

- IfLT: If Logical True

  Instruction → Integer↑r Label↑n

  EMIT↓'IfLT'↓r↓n
  EMIT↓'Nop'
  EMIT↓'Nop'

- IfLF: If Logical False

  Instruction → ¬Integer↑r Label↑n

  EMIT↓'IfLF'↓r↓n
  EMIT↓'Nop'
  EMIT↓'Nop'

8. Appendix B: U-code to IR Translator Design

This section describes the design of the translator in detail. It is written to serve as a summary program documentation for people who may want to modify/extend the translator. The references to system itnplcmcntation details (e.g., file names) can be ignored by the casual reader.

8.1 Design

The U-code to IR translator is written entirely in DECSYSTEM-20 Pascal (i.e., Hedrick Pascal). The non-standard Pascal features used have been limited to a few system features (QUIT, RUNTIME, TTY, random file access) that should make this translator simple to transport to other Pascal implementations. No non-standard data or control structures have been used. The translator consists of 8 separately compilable modules that are described later. The following naming conventions have been used for file name extensions:

- .PAS - a .PAS (Pascal) file contains the program text of a module.
- .IMP - a .IMP (Import) file contains EXTERN procedure & function declarations of the module that can be INCLUDEd[61] by other modules.
- .INC - a .INC file contains CONST and TYPE declarations that can also be INCLUDEd by other modules.

The physical specifications of the translator are as follows:

Number of modules = 8

Total program text size = 84 pages = 210K chars ≈ SK lines

Total executable program size = 69 pages = 34.5K words

Average execution speed ≈ 500 U-code instructions/CPU second

The translator has 8 modules that are described below, along with the names of the files that contain them:

Main program - UCTOIR.PAS

This program is the main module of the translator. It calls all the required file initialization routines, and then hands over control to the parser for the rest of the program execution. At the end, it displays some job statistics of the translation performed - the number of U-code instructions processed and the amount of CPU time spent on the translation.

---

[61] The INCLUDE statement in DECSYSTEM-20 Pascal is used to insert the contents of the specified file at that point.

U-code initialization - UINI.IMP, UINI.PAS

This module has a global array, *Utabrec* that stores the following information for each U-code instruction:

- Opcode name
- Instruction length
- Instruction format

This information is then used by the U-code rcadcr. The array is initialized by a call to procedure *Uini* from the main program. This module is directly available in the U-code system, and has not been modified in our translator.

U-code reader - BREAD.INC, BREAD.IMP, BREAD.PAS

Readers for both U-code and B-code (binary U-code) are available in the U-code system. Our translator uses the B-code reader, as binary U-code requires less space and can be read faster. This module has an input procedure (ReadUinstr) that reads one U-code instruction at a time and returns an encoding of the instruction in a Pascal record (TYPE Bcrec).

Symbol table reader - D.INC, DREAD.IMP, DREAD.PAS

This module is an adaptation of the Symbol table reader, DREAD (Debugger READ) available in the U-code system debugger. The debugger uses this module to parse the U-code symbol table file. Our translator also uses it for the same purpose. It was necessary to make some modifications because the debugger also used a random access data structure, initialized elsewhere, that was not used in our translator.

Symbol table module - SYMTAB.IMP, SYMTAB.PAS

Our translator requires more sophisticated symbol table access than just reading a procedure's declarations sequentially. This facility is required because IR has block structure like Pascal, and so all outer declarations have to be printed before a procedure's own declarations. Further, parameters have to be identified and declared in the callee, in the same order as they occur in the caller. The U-code symbol table entries are sorted alphabetically, which destroys the ordering of parameters. Thus, the U-code PSTR instructions have to be used to identify the correct order. This module exports PROCEDURE Getproctablc that does all this processing required to print the procedure's declarations correctly.

U-code parser - PARSECONSTS.INC, PARSE.INC, PARSE.IMP, PARSE.PAS

An automatic parser, generated by an LALR(1) parser generator [LALR], is used to parse the input U-code. It is not sufficient to examine each U-code instruction indcpcndcntly as is done in code gcncrators. For translation to IR, it becomes necessary to identify structured scqucnccs of U-code instructions, e.g., a

sequence of stack operations represents an arithmetic expression. The actual translation is achieved by semantic routines that need to be called at correct instants in the parse.

Semantic Routines - ACTION.INC, ACTION.IMP, ACTION.PAS

Semantic routines may be associated with either of these two parsing actions:

1. shift - This action normally has a limited use, because the corresponding semantic action must be performed, whenever the terminal is encountered, irrespective of context. We have used it for the MST and ENT U-code instructions.

2. reduce - This action is the conventional technique of incorporating semantic actions into bottom-up parsing. The production used for reduction determines the semantic routine to be called.

All the semantic routines communicate with each other through the *semantic stack.* This stack is *polymorphic,* and a stack entry is essentially one of:

1. An expression tree - the semantic record for the *non-terminal* expression that is the only *non-terminal* for which any information needs to be stored.

2. A U-code instruction - the semantic record for any *terminal* that needs to be stored, since all *terminals* are U-code instructions.

Each semantic routine uses a certain number of stack entries from the top, pops those entries when it is done, and possibly pushes a new entry, along with generating appropriate IR code. This scheme is therefore ensured to work for semantic routines called by nested syntactic structures.

IR writer - IRWRIT.IMP, IRWRIT.PAS

This module controls the generation of the output IR program. All other modules use its routines to finally output the translation.

Files used

The following is the list of files that make up the U-code to IR translator; the number of lines of source code is listed in parentheses alongside:

ACTION.INC	-	CONST/TYPE declarations defined in module ACTION. (52 lines)
ACTION.IMP	-	Procedures exported by module ACTION. (37 lines)
ACTION.PAS	-	Module ACTION. (933 lines)
ACTION.REL	-	Compiled, binary, relocatable code.
BREAD.INC	-	CONST/TYPE declarations defined in module BREAD. (6 lines)
BREAD.IMP	-	Procedures exported by module BREAD. (11 lines)
BREAD.PAS	-	Module BREAD. (644 lines)
BREAD.REL	-	Compiled, binary, relocatable code.
D.INC		CONST/TYPE declarations defined in module DREAD. (119 lines)

DREAD.IMP	-	Procedures exported by module DREAD. (9 lines)
DREAD.PAS	-	Module DREAD. (638 lines)
DREAD.REL	-	Compiled, binary, relocatable code.
IRWRIT.IMP	-	Proccdurcs exported by module IRWRIT. (88 lines)
IRWRIT.PAS	-	Module IRWRIT. (1224 lines)
IRWRIT.REL	-	Compiled. binary, relocatable code.
PARSEFILE	-	Automatically generated parsing tables used by the LALR(l) U-code parser.
PARSECON STS.INC	-	CONST declarations for production and grammar symbol numbers in the parser. (30 lines)
PARSE.INC	-	CONST/TYPE declarations defined in module PARSE. (13 lines)
PARSE.IMP		Procedures exported by module PARSE. (10 lines)
PARSE.PAS	-	Module PARSE. (505 lines)
PARSE.REL	-	Compiled, binary, relocatable code.
SYMTAB.IMP	-	Procedures exported by module SYMTAB. (7 lines)
SYMTAB.PAS	-	Module SYMTAB. (251 lines)
SYMTAB.REL	-	Compiled, binary, relocatable code.
UCODE.INC	-	CONST/TYPE declarations in the U-code system.
USYS.FOM		CONST/TYPE declarations for FOM.[62]
UCTOIR.PAS	-	Module UCTOIR - the main program. (100 lines)
UCTOIR.REL	-	Compiled, binary, relocatable code.
UCTOIR.EXE	-	Executable code for the entire translator obtained entirely from compiling the 8 modules, and linking their .REL files.
UINI.IMP		Procedures exported by module UINI. (6 lines)
UINI.PAS		Module UINI. (327 lines)
UINI.REL		Compiled, binary, relocatable code.

8.2 U-code to IR translator User Manual

This section describes how to use the U-code to IR translator. The translator takes as input a B-code (binary U-code) program file (say X.BCO) and its symbol table file (say X.SYM). and generates IR code (in file X.IR, say) as output. The file, UCTOIR.EXE, contains the executable version of the translator. It may be invoked as shown below (characters to bc typed in by the user are shown in bold face):

@RUn UCTOIR
BCODE-IN : X.BCO
SYMBOLTABL : X.SYM
IR – OUT : X.IR
FARSEFILE : PARSEFILE

---

[62] This table is the machine-dependent table of constants used to generate U-code for a particular machine. It is needed by the translator to correctly convert the machine-dcpendcnr sizes and offsets to machine-independent values in IR

```
Echo U-code instructions in IR program? (y/n) [Default:n] :

Start execution of U-code to IR translator . . .

End execution.
Runtime = 2934 milliseconds.
Number of U-code instructions processed = 1654
Average processing rate =  550.2 instructions/second.
@
```

PARSEFILE contains the parsing tables to be used by the LALR(l) parser for U-code. The option of echoing U-code instructions as comments in IR code is useful for comparing the two intermediate languages. It is also a useful debugging/development aid for further extensions to the translator. As indicated above, the option is invoked by typing a 'y' in response to the question. Anything else will disable the option.[63] The translator will abort execution in an error situation, e.g., if an unsupported feature is present in the U-code program. In this case, the "End execution" message will not be displayed on the terminal. Error messages appear at the end of the IR ouput file, and are easily recognized as they are enclosed within two '****' strings. If the error message refers to an unsupported feature, then that feature should be removed from the original HLL program, which should then be re-compiled.

Fifes belonging to the translator system

All files that belong to the translator are outlined in the preceding section. There is one more file that helps identify missing files. It is called FILES.CTL, and it contains a DIRECTORY command with the list of all the files used by the translator. It can be invoked by typing

```
@DO FILES.CTL
```

The DIRECTORY command will automatically notify the user of any missing files.

[63] On TOPS-20, you need to hit <return> twice, if you do not type any character.