THE CODE GENERATOR GENERATORS' WORK STATION: REIMPLEMENTATION AND EXPERIMENTATION WITH THE GRAHAM-GLANVILLE MACHINE INDEPENDENT ALGORITHMS FOR CODE GENERATION

by

Robert R. Henry

Memorandum No. UCB/ERL M81/47

4 June 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

This work was supported by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144.

Table of Contents

Figures

1. Introduction

In December 1977, R.S. Glanville completed his doctoral dissertation describing a machine independent algorithm for code generation and its use in retargetable compilers. The first part of Glanville's algorithm, called the code generator generator, takes a description of the target computer for which one wants to generate code, and produces a set of tables describing the target computer, relying heavily on LR parser construction technology. This part of the algorithm is run once per machine description. The second part of his algorithm reads an intermediate representation (IR) of a program being compiled into code for the target computer, interprets the tables produced by the first algorithm and maps the intermediate representation into the instruction set of the target machine. The second part of the algorithm is run once per source program. The mapping is provably correct, and produces nearly optimal code at the expression level. The reader is assumed to be familiar with Glanville's dissertation, and associated papers [Glan77] [GlanGrah78] [GrahGlan78] [Grah80].

For readers familiar with Glanville's dissertation and the work he did in its support, this report documents in detail what was done to improve or extend his dissertation work. We reimplemented Glanville's basic algorithms, and implemented a hospitable environment to experiment with retargetable code generators using Glanville's ideas. The reimplementation and environment is collectively called the Code Generator Generator's Work Station, or CGGWS. It is designed to be run on a 32 bit machine running UNIX[1]. Some of Glanville's algorithms were changed: his loop detection algorithm was combined with the table generator, as well as with the default list constructor. The former combination made the implementation easier, and the latter corrected a serious omission in the dissertation. We implemented a simple compression technique for the large parse tables. However, experience with many real machines and complete machine descriptions shows that the description tables, in their original design, are too large to be practical. In addition, we implemented a code generator which interprets the description tables, and which is capable of generating reasonably efficient assembly code for arbitrarily complicated expressions for a wide variety of target machine architectures.

This report is also intended to provide the rationale behind the implementation. This rationale is missing from the code implementing CGGWS. We describe the quantitative results of the implementation, encapsulate the experience gained writing machine descriptions for a number of machines, and offer cook-book suggestions for writing machine descriptions. Throughout the report, we explain the problems with this code generation technique in its current form and the issues that we are examining in our current research in machine independent code generation.

[1]UNIX is a trademark of Bell Laboratories.

2. Implementation Overview

The implementation description first briefly describes a previous implementation, then discusses the current implementation as a whole by describing its five phases, its inter phase communication files, its coding conventions, and finally the major data structures that the code generator generator constructs for the code generator. Then, each phase is sequentially examined in order of application, to describe how that phase contributes to the major data structures. There are extensive descriptions of the second phase (table constructor) and the fifth phase (code generator) which document changes to Glanville's algorithms, and discuss semantic issues mentioned only briefly in his dissertation.

2.1. Glanville's Implementation

As part of his dissertation research, Glanville implemented his algorithms as a series of PASCAL programs. He chose PASCAL because of its control and data structuring capabilities, its set data types and algebra, and because the compiler and interpreter had superior error diagnostics. However, the Berkeley PASCAL he used ran interpretively on a PDP11[2], was very slow, and placed severe constraints on the amount of data and code his program could have. Consequently, the target machine descriptions he considered were restricted to small instruction subsets of real machines. (The PDP11 and IBM360 subsets given in his dissertation are the largest examples that could be handled by his implementation.) PASCAL as a language does not allow dynamic arrays, which are mandatory for flexible and efficient implementation of the code generator. The implementation itself was a research program; some of the code was obscured by the development history; all of the code was very dense, comment free, and hard to understand. It also possessed latent bugs not demonstrated by his dissertation examples. In addition, both implementation space constraints, as well as artifacts from the order in which the ideas were developed, forced the algorithmic separation presented in his dissertation. Further, his dissertation and implementation algorithms disagreed in some places. However, Glanville's programs were sufficient to show that his ideas worked.

2.2. Henry's Reimplementation

With some of the deficiencies in the PASCAL implementation in mind, the algorithms were reimplemented in the language "C". There were three goals of the reimplementation. First, reimplementing in almost any language would force the author to understand the algorithms thoroughly, possibly suggesting some basic improvements or modifications. Secondly, the new implementation would be cleaned up enough for serious production work. Thirdly, an implementation in "C" could exploit some of the power in that language. "C" has most of the data and program structuring facilities available in PASCAL, generates compiled code, and provides low level constructs allowing efficient set implementations and memory management. "C" provides facilities to allocate arrays dynamically, provides an operating system interface facilitating efficient I/O, and is directly useable with the tools *lex* and *yacc*. In

[2]PDP is a trademark of Digital Equipment Corporation.

addition, when the project started, Berkeley had just purchased a VAX[3] 11/780 having a "C" compiler, but the Berkeley PASCAL system had not yet been ported.

For comparision's sake, Glanville's original implementation contained 3486 lines of PASCAL. For the reasons discussed in § 2.1 and in this section, this implementation was entirely discarded. William Joy began the implementation of CGGWS, and wrote about 2000 lines of "C" code implementing the first part of CGGWS. The author completed the implementation, adding approximately 22,000 lines of "C" over the course of 2 years.

The reimplementation has failed to meet all of its goals. The author did not initially know "C", so some of the first modules to be written exhibit rough edges and clumsiness still persists. Parts of the code generator are obtuse because the register model and register allocation schemes were hard to master. Algorithmic, logical, time, efficiency and development constraints resulted in 89 partially shared source files implementing the four phases. (See appendix 6.) Fortunately, the program *make* provided a reasonable environment to manage the source. Further, the interphase file structure is more confusing than that with which Glanville's test implementation was saddled. Local comments are liberally inserted into the code, but the code lacks global documentation. This report attempts to supply that picture.

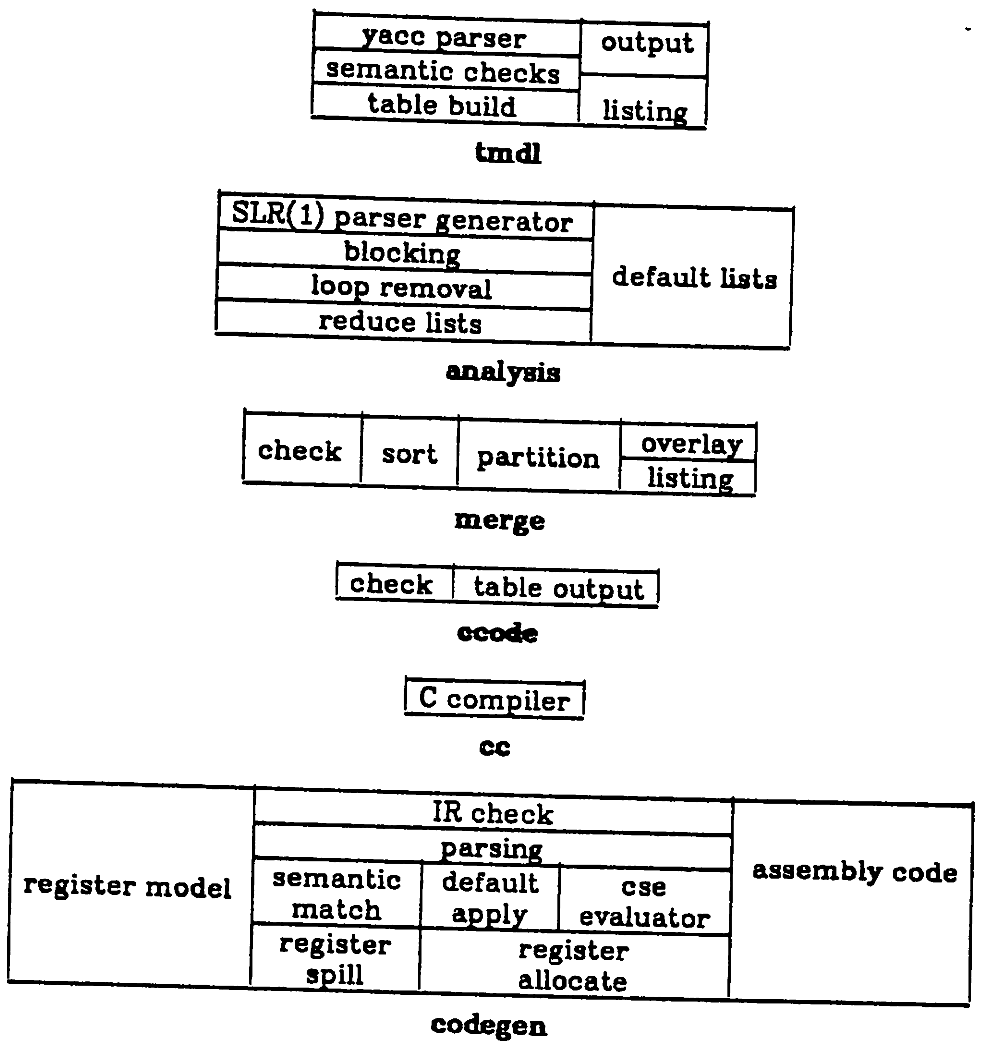
2.3. Implementation Phase Structure

The implementation is structured into five physical phases. Figure 2.1 shows the tasks allocated amongst the five phases. Its form is based on the diagrams from the Bliss book [Wulf75]. The first four phases are collectively called the code generator generator, or parser constructor. The phases are run sequentially once per machine description to construct the code generator tables. They communicate via a number of interphase temporary files. The last phase is run once per source program. **Tmdl** syntactically and semantically analyzes the target machine description language (TMDL) described in Glanville's dissertation. **Analysis** analyzes the grammar, and implements the SLR(1) parser constructor, loop removal, blocking analysis and default list construction also described in his dissertation, eventually producing a number of tables describing the machine. **Merge** merges the output files produced by the first two phases, compresses the tables and does consistency checks. **Ccode** converts the internal tables massaged by **merge** into initialized C variables that are then compiled and loaded in read only storage together with a set of standard modules that implement **codegen**. The C Compiler is not counted as a phase of CGGWS. **Codegen** does all of the IR semantic checking and register allocation.

2.3.1. Typesetting Conventions for this Report

Phases in CGGWS appear in bold, viz **tmdl**, **analysis**. Utility programs appear in italics, viz *lex*, *yacc*. The abbreviation TMDL, in roman, stands for Target Machine Description Language, and is not to be confused with the compiler for TMDL, **tmdl**. Reserved words in TMDL appear in bold, viz **$registers**, **$symbols**. C source files implementing CGGWS appear underlined, viz <u>seman.h</u>. Variables in source files are in italics, viz *syms*, *rules*. Algorithm names appear in italics, as *expandR.45*.

[3]VAX is a trademark of Digital Equipment Corporation



CGGWS flow goes from top to bottom, left to right within each phase diagram.

Figure 2.1: Functional Overview to the CGGWS

Finally, standard typesetting conventions are used for grammar terminology, such as the sets **N**, **P**.

2.3.2. Data Structure Terminology

We describe here the data structure terminology used throughout the report. A logical or virtual data structure represents the high level visualization of the data on which an algorithm works. Both the algorithms in Glanville's dissertation, and those presented later for **analysis**, deal exclusively with logical data structures. Logical data structures are the same, regardless of the phase. Physical data structures are the physical representation of the logical structure, are implemented to trade off speed versus time, and allow speedy responses to the algorithmic requirements. Physical data structures are explicit in the C structure and variable declarations, are implicit in the code, and vary considerably from phase to phase. If a datum *indexes* an element of **A**, then **A** is represented as an array of contiguously allocated elements. An indexed structure is space efficient, but requires a multiplication to address the indexed element and is not extendable once allocated. Indices are phase independent. If a datum *points* to an element of **A**, then the element in **A** is directly addressed, requiring slightly more space than an index's implicit address, but access is extremely fast, and is phase dependent. *Binding* converts indices to pointers; *unbinding* converts pointers to indices, although the index may not be valid in the current phase.

We list here some other abbreviations used throughout this report that also appear in Glanville's dissertation. More extensive definitions will be found in other sections of this report.

IR	*I*ntermediate *R*epresentation (of the program)
canon	*Canon*icalized (IR)
cse	*C*ommon *S*ub *E*xpression
lhs	*L*eft *H*and *S*ide (of a grammar rule)
rhs	*R*ight *H*and *S*ide (of a grammar rule)
rlist	*R*educe *List*
dlist	*D*efault *List*
proc	*Proc*edure

2.3.3. Phase Subdivision Rationale

Tasks are allocated to the first four passes in a way to satisfy both data structure space and time constraints. The phase structure also reflects the way the programs were developed. These two constraints will be discussed in turn.

2.3.3.1. Data Structuring Constraints

If a phase generates a dynamically unpredictable number of elements in a logical data structure, each element is allocated individually, and the elements are linked together with pointers to form the physical structure. The output routine for that phase concatenates the elements together contiguously into their proper order when the routine writes to the interphase file. The next phase knows the number of elements and size of each element, so it can dynamically allocate the entire array, and do a single, efficient interphase file read. (The above dynamic allocation is not strictly true for the **tmdl** phase. **Tmdl** has large upper bounds on the number of register definitions and the size of a grammar rule.)

The entire code generator generation task is broken into phases at points where as much dynamic information as possible has been assembled and the local auxiliary data structures are no longer needed. At this point, the physical structure of the major virtual data structures is too cumbersome for the future use and can benefit by the reorganization possible across phases. For example, **analysis** does itemset manipulation that no other phase needs, but does not need to see the entire parser at once. **Merge** needs to see the entire parser to compress the tables, but does not need some of the semantic information associated with rules.

A major and unavoidable difficulty with the separate phase structure and phase specific data representations is the plethora of structure definitions and the confusingly different code in the different phases.

2.3.3.2. Software Development Constraints

The first two code generator generator phases, **tmdl** and **analysis** were written with the intent that **codegen** would be the following phase. All of **tmdl**, plus the SLR(1) parser constructor in **analysis**, was written by Bill Joy before the author took over the project. Hence, the established coding style, variable names and other conventions had to be learned and observed before default and reduce list construction could be added to **analysis**. Even though a general strategy was known before and during **analysis**'s implementation, the implementation agglomerated piecemeal as more algorithms were added and tested.

When **analysis** was completed, and the framework for **codegen** laid out and its implementation begun, it became apparent that the interphase file structure between **analysis** and **codegen** was too bulky, and the machine description tables could be compressed. What followed was entrapment in the quagmire of a large system. By that time, **analysis** was too large and complicated for the author to regain the familiarity present during its intense development. Modifying **analysis** would have created more problems than it would have solved; touching one part of the program might inadvertently touch and break another fragile part of the program.

Hence, **merge** came into being. **Merge** could benefit from the author's improved C coding style and could establish its own tailored and conceptually simplified environment in which to complete the task of table compression. For the same complexity reasons, **ccode** was written as a separate phase, once **merge** and **codegen** had been working together for 6 months. Adding the two phases was not without problems. The interphase communication file had to reflect the internal specifics of both communicating phases. As a result, physically clean data structures have murky starts in their interface initialization routines. Writing and testing a new phase was extremely difficult because *two* interphase file protocols and implementor had to be modified and understood. A bug in **merge**'s input routine might require modification to **tmdl**, **analysis** and **merge**, plus a run of the standardized test example through the modified phases, before **merge** could be retested.

Ccode had its own development difficulties. That phase had to produce output syntactically and "micro" semantically acceptable to the C compiler, plus "macro" semantically acceptable to **codegen**. Since any modifications to structure definitions across all of CGGWS must change code in **ccode**, **ccode** was written after all other phases were frozen.

The five programs, developed over a period of 1.5 years, were quite sensitive to changes in the development environment. Modifications to both the semantics of C, and to the compiler and optimizer, proved to be a continual nuisance. Language shortcuts and machine specifics accidently added to the source code of the five phases were invariably stepped on and broken by compiler modifications.

2.4. CGGWS Test Cases

2.4.1. Machine Description Grammars

CGGWS was tested on a repertoire of eight machine grammars. With one exception, these grammars describe just the basic instruction set of the target machine, for integer typed operators and operands only. The grammars recognize IR trees constructed from the IR operators and operands Glanville used for his PDP11 and IBM360 examples. A few grammars have been augmented to recognize procedure and function call, return and argument passing; others have been extended to recognize productions creating and using common sub expressions. For all machine descriptions, idiomatic instructions are restricted to clear, increment, decrement, shift left to multiply and test against zero. For all but one machine grammar, constant operands to commutative plus and multiply IR operators are assumed to be *canonicalized* by the IR generator so they are in the left subtree. This reordering saves a great deal of space in the resulting code generator. With the exception of the HP3000, all machines are classical Von Neumann general register set architectures, so it was relatively tedious and repetitious to write the description grammars. The size of the code generators produced by some of the machine description grammars is in figure 6.1 of §6; values in figure 6.1 are one measure of target machine complexity. Figure 2.2 describes the machine grammars.

machine	*name*	*cses?*	*procs?*	*canon?*	*notes*
PDP11	glan11	no	no	yes	Incomplete grammar. Incorrect double register specifications. This is only included for comparison of CGGWS with the results in the dissertation.
PDP11	11	yes	yes	yes	Grammar complete for integer data types only. Partial attempt to handle **bic** instruction.
PDP11	full11	yes	yes	yes	Byte, integer and floating data types (no double). A code generator from this grammar was never used to generate code because of its unwieldy size.
PDP11	nc11	yes	yes	no	Non canonicalized version of 11 grammar, above.
IBM360	glan360	no	no	yes	Incomplete grammar. This is only included for comparisons of CGGWS with the results in the dissertation.
Z8000	Z8000	no	no	yes	Machine description grammar developed early in the CGGWS history. Quite incomplete.
8086	8086	no	no	yes	
HP3000	3000	no	no	yes	Stack machine; top locations of the stack treated as a register file.
VAX11	VAX	no	no	yes	**movl** grammar: all possible canonicalized operands to the integer move instruction, plus register to register add and multiply so default lists can be constructed. No idioms. Included to show enormous size of CGGWS grammar, even with canonicalized input.
Gates	NAND	no	no	no	Hardware description language when only NAND gates are available to implement combinatorial logic. Includes DeMorgan's rule as an idiom, plus other axioms of Boolean algebra. Never used for any serious work.

Figure 2.2: Facts about Ten Machine Description Grammars

2.4.2. IR Programs

The majority of CGGWS was debugged with the hand-generated IR for the matrix mutliply program found in Glanville's dissertation (figure 5.8, page 96), or from modified versions of that basic algorithm. When **codegen** was written and debugged, interesting IR test cases were given to **codegen** interactively to exercise the buggy portions. Later in the development cycle, **IRgen** was written to generate IR automatically from a language reminiscent of "C"; see §9.2.

3. Major Data Structures

This section briefly describes the data structures the code generator generator constructs for the code generator. Interspersed with the description are references to particular C source language files. A complete description of the source files may be found in Appendix 6.

A number of minor control structures, and structures describing the target machine, are declared in seman.h and are constructed by **tmdl**. The action, next, reduce and default tables are constructed by **analysis** and physically manipulated (but virtually unchanged) by **merge** and **ccode**; these tables are declared in tabledef.h.

3.1. Data Derived from the Machine Description

3.1.1. Options

Debugging print routines in all phases are enabled by setting boolean, numeric, or string valued options. These options are set either by the process arguments passed at the shell command level to the physical phase, or from the **$options** section in the TMDL source. Options set in the **tmdl** phase are propagated through all subsequent phases.

Data structures describing the target machine have a natural and obvious correspondence to the semantics of TMDL. (§3.3 of Glanville's dissertation describes the semantics of TMDL.) These structures describe the registers, the symbols making up the instruction grammar, and the syntax and semantics of the instruction grammar. During **analysis** and **merge**, some semantic attributes for both symbols and grammar rules are not memory resident, or are kept resident in an abbreviated form.

3.1.2. Registers

All registers are enumerated in the TMDL **$register** section, and saved in the array *regnames*. Allocatable registers are managed by the code generator, and may not appear as register qualifications in the IR. Dedicated registers may appear in the IR provided they are bound to a non terminal register symbol, in which case they are checked only for semantic correctness, and will not be allocated by **codegen** for expression temporaries.

3.1.3. Symbols

TMDL symbols are syntactically either grammar non terminals or grammar terminals, and semantically are either registers with members from the register set, register pairs with paired members from the register set, unary or binary prefix operators, or ranged constants [4]. Symbol attributes vary depending on their semantics, and are represented as union elements in the array *syms*. Implicit in the **$symbol** section of TMDL is the dot symbol (λ), overloaded to represent both no result location and the end of the IR file (conventionally, "**$**"). Dot is treated

[4] Note that we have assumed in writing these programs that all non terminals would be registers. That assumption is not generally valid. For example, in the VAX, memory locations could be used as accumulators.

uniformly as a symbol from **N** in all phases of the code generator generator. Symbol enumeration order follows almost the same conventions in the first phases, but due to the reordering **merge** does, the enumeration order is different for **codegen**. In the first three phases, dot is the zeroth symbol, and symbols with ordinal position less than the number of grammar non terminals are grammatical non-terminals; otherwise they are terminals. Macros in seman.h provide symbol iterators and discrimination, and are internally messy, but easy to use. It was difficult to keep the symbol enumeration consistent across phases during development because of differing conventions, the messy interphase I/O, and conventions unique to **tmdl**; these all resulted in a maintenance nightmare.

3.1.4. Rules

Rules are implemented as elements of a vector. Each rule has four macroscopic parts, in a one to one correspondence to the parts of a rule definition in the TMDL **$rule** section.

3.1.4.1. Left and Right Hand Side

Both the left hand side (*lhs*) and right hand side (*rhs*) of a rule are vectors of *bundles*. A bundle describes that environment for that symbol within the rule, including the semantic qualifications and their associated value, as described in Glanville's dissertation in §3.2. The bundle contains a unique sequence number shared amongst all bundles with the same qualification and linked into the assembly field, and an index to the next bundle with the same unique sequence number. Both of these fields help with the semantic checking and symbolic substitution done by the code generator.

There are five different ways to qualify symbols semantically. Two of these are unique for double registers; these will be described later. *Dot* qualifications, represented by "." in TMDL, are inter bundle links, or link a bundle into the assembly field. *Equal* qualifications, represented by "=" in TMDL, are requests for specific values, depending on the kind of symbol they are qualifying. The third kind of qualification is no qualification at all; in this implementation of CGGWS, unary or binary operators must not be qualified. Throughout this report, qualifications will be referred to by either their name (*dot* or *equal*), or by their TMDL representation ("." or "="), with corresponding conventions for double register qualifications, introduced in §3.2.

Taken together, all bundle qualifications determine how semantically restricted that rule is. When the code generator parser makes a reduction, it attempts to select rules with the largest semantic restriction value, progressing on failure to weakly restricted rules, and eventually to an unrestricted rule or a default list. Rules with high restriction values usually describe more efficient instructions on the target machine. According to our ordering, rules with non terminal bundles having "=" qualifications possess the highest restriction level, since a specific register or constant is requested. Figure 3.1 gives our ordering, in order from high to low restriction. A rule may simultaneously satisfy more than one of these restrictions.

restricted?	*case*	*Notes*
	1	Identical *equal* qualification between the left hand and right hand side of the rule. The destination register is the same as one of the source registers.
yes	2	*Equal* qualification on the left hand side of a rule; The destination is a specific register.
yes	3	*Equal* qualification on the right hand side of a rule (A specific register or constant.)
	4	*Dot* qualification between the left and right hand side (A register used on the right hand side will hold the result on the left hand side.)
yes	5	*Dot* qualification between bundles on the right hand side.
	6	No restriction at all.

Figure 3.1: Semantic Restriction Levels

Case (1) (figure 3.1) is a combination of cases (2) and (3). The semantic attributes recognized by case (1) are mutually exclusive from those recognized by cases (2) and (3), although if there are other "=" bundle qualifications in the production, cases (2) and (3) may still be set. For all machine grammars we have studied, productions have case (1) semantic restrictions only for their register allocation side effects, and in general can not be decomposed by the default list constructor in **analysis**.

For example, on the PDP11, the chain rules

```
r=r0 → e=r0
r=r2 → e=r2
```

convert registers classed as even registers to the general register class. Or, in the extended and corrected grammar for the PDP 11 (see §3.2 and figure 3.2 for complete discussion), the four productions

```
d=r0 → * k.1 dprod=r0
d=r2 → * k.1 dprod=r2
d=r0 → * dprod=r0 k.1
d=r2 → * dprod=r2 k.1
```

model the instruction multiplying an integer held in a double register pair by a scalar constant. Two different non terminals, *dprod* and *d* are needed because they represent register pairs with different internal layouts. Further, an "=" between the left hand and right hand sides is needed to specify the semantic link between the two different non terminals, as a "." link may only be between identical symbols. Since no default list can be constructed from the first case (there are no shorter rules; see §5.1.14), it is assumed that the TMDL grammar has enumerated all the possible productions with case (1) semantics to ensure that semantic blocking is impossible. This assumption is not checked in the TMDL.

The distinction and ordering between cases (4) and (6) helps the code generator choose instructions minimizing register allocation side effects. For example, these two rules model the VAX's long integer register to register add instruction:

```
r.1  →  + r.2 r.3      "addl3 r.2, r.3, r.1 "
r.3  →  + r.2 r.3      "addl2 r.2, r.3 "
```

Normally, both rules would be considered semantically unrestricted, but the first rule with three operands should be applied by **codegen** only when the value r.3 has future uses, as the three operand instruction is less efficient. The ordering between cases (4) and (6) ensures that the two operand instruction is tried first.

If a rule is marked as restricted in Figure 3.1, then **analysis** considers it to be semantically restricted. This may force **analysis** to construct a default list (see §5.1.14).

3.1.4.2. Assembly String

This string is a massaged TMDL assembly string, with references into the rule's bundles.

3.1.4.3. Rule Qualifications

Qualification values include: The cost of the rule (either space or time, or both), a flag indicating that applying the rule has a "magic" register allocation side effect (used to process common sub expressions), and the semantic qualification level. Both the cost and magicness of a rule are specified in rule qualifiers in the TMDL. The cost is not currently used by the code generator.

3.2. Double Registers

Some target machines have instructions producing or consuming operands in *double* registers. A double register is composed of two other general, single length registers termed *siblings*. For most older machines, the only available form of multiply or divide uses double registers, which for **codegen**, as for other code generators, are difficult for the register allocator to handle semantically because of their size and alignment constraints. Like other register classes, double registers are declared in the TMDL **$variables** section, with their specific constituent elements paired into a first and a second sibling by "<" and ">" angle brackets. An oversight in the implementation of **codegen**'s general register model requires that a single register always have the same sibling, regardless of the number of double register variables of which it is a member. This model is adequate for *even-odd* paired machines like the PDP11 or IBM360 (register pairs (r0, r1), (r2, r3), (r4, r5)), but it can not fully model *adjacent* paired machines like the VAX, Z8000 or HP3000 (register pairs (r0, r1), (r1, r2), (r2, r3), etc). To model the adjacent paired registers in our implementation, adjacent pairs must be discarded until an even-odd paired machine remains.

Double registers may be semantically qualified in four different ways. A double register may be broken down into its constituent members either for printing or for *value decomposition*, or the double register may be treated as an indecomposable entity just like single registers always are. Note that Glanville's definition for TMDL only allowed three kinds of semantic qualifications for double registers, but he was unable to do value decomposition unambiguously. In the following discussion of the four qualification methods, refer to figure 3.2, which contains the interesting subset of the double register grammar for the PDP11. We assume that *d* is a double register, *r* is a single register, both *m* and *n* are integers, and both *xx* and *yy* are allowable names for the first and second siblings of *d*, respectively.

d.m	[indecomposable] This is an inter bundle dot qualifier, used in the same way as dot qualifiers on single registers.
d=xx	[indecomposable] This is an explicit register requested via the equal qualifier. The requested register, *xx*, *must* uniquely refer to the first sibling register of a double register pair.
d.m .n	[decomposable] This is a *double dot* qualification on the double register *d*. Double dot qualifications may only appear in the assembly string, and are the only way double registers appearing in the assembly string can be qualified. Here, *m* is an inter bundle dot qualifier, and *n* is either 1 or 2, specifying the print name for the first or second sibling.
r<d.m .n>	[decomposable] This is a *angle bracket double dot* qualification. It may appear only on the left hand side of a production, and decomposes a double register pair into one single register that is retained; the other single register is discarded. This qualification form is discussed in the next paragraph.

As an example of the bracket double dot qualification, consider the divide instruction on the PDP11. For that machine, the dividend must initially be in a double register pair. That register pair is also used as the destination. As the destination, the pair will contain the quotient in the first (even) sibling, and the remainder in the second (odd) sibling. For a given divisor, the hardware divide instruction is modeled in TMDL as two different productions (here we show it with a divisor of a scalar constant):

```
e<d.1 .1> → / d.1 k.2   "div k.2, d.1.1 "
o<d.1 .2> → % d.1 k.2   "div k.2, d.1.2 "
```

(Here, we see three different kinds of double register qualifications in the same production.) Even though both a quotient and a remainder are produced, only one of the two values can be tracked by TMDL and **codegen**'s register model. The other value must be discarded. The saved value is to be considered as a single register in the class "e" (even) or "o" (odd), as specified by the non terminal appearing outside the angle brackets on the left hand side of the productions.

```
$symbols
    $variables
    r=  r0, r1, r2, r3, r4, r5, sp, pc;
    e=  r0, r2;              /*even registers*/
    o=  r1, r3;              /*odd registers*/
/*
 *d: a true double length register, with the low order word (LOW)
 *in the odd reg, and the high order word (HOW) in the even reg.
 *dprod: a double register allocated just to have two adjacent registers,
 *but containing just a single length result. The even register contains only
 *low order word; the odd register contains junk.
 */
    d=  <r0, r1>, <r2, r3>;
    dprod = <r0, r1>, <r2, r3>;
$instructions
/*
 *Formation of a double length result in preparation for a divide
 */
    d=r0  → o=r1        "tst r10sxt r0; div setup", cost=2;
    d=r0  → e=r0        "mov r0, r10sxt r0; div setup", cost=2;
    d=r2  → o=r3        "tst r30sxt r2; div setup" cost=2;
    d=r2  → e=r2        "mov r2, r30sxt r2; div setup" cost=2;
/*
 *Formation of a double length result in preparation for a multiply
 */
    dprod=r0  → o=r1  "mov r1, r0; mul setup", cost=1;
    dprod=r0  → e=r0  "; mul setup", cost = 0;
    dprod=r2  → o=r3  "mov r3, r2; mul setup", cost=1;
    dprod=r2  → e=r2  "; mul setup", cost=0;
/*
 *Extracting the result of a multiply. Extracting the result of a divide
 *is done automatically by the <..> qualification on divide and remainder.
 *We don't use the same method for multiply, because a divide may follow
 *immediately after the multiply, and there is no point is disassembling
 *a register, and then immediately reassembling it.
 */
    o=r1  → d=r0        "; discard HOW of d=r0", cost = 0;
    o=r3  → d=r2        "; discard HOW of d=r2", cost = 0;
    o.1  → * o.1 k.2 "mul $k.2 , o.1" cost = 2;
    o.1  → * k.2 o.1      "mul $k.2 , o.1" cost = 2;
    d=r0  → * dprod=r0 k.1 "mul $k.1 , r0 " cost = 2;
    d=r2  → * dprod=r2 k.1 "mul $k.1 , r2 " cost = 2;
    d=r0  → * k.1 dprod=r0 "mul $k.1 , r0 " cost = 2;
    d=r2  → * k.1 dprod=r2 "mul $k.1 , r2 " cost = 2;
    e<d.1 .1>  → / d.1 k.2   "div $k.2 , d.1.1 " cost = 2;
    o<d.1 .2>  → % d.1 k.3   "div $k.3 , d.1.2 " cost = 2;
/*
 *Other productions for multiply and divide follow.
 */
$end
```

Figure 3.2: Double Register Grammar for the PDP11

The TMDL production for divide

e<d.1 .1> → / d.1 k.2 "div k.2, d.1.1 "

is syntactically equivalent to:

e.1 → / d.1 k.2

throughout **analysis**. However, to the register allocator in **codegen**, the production is semantically equivalent to

d.1 → / d.1 k.2

with the left hand side to right hand side dot binding on "d" ensuring that the same double register is allocated for both the source and the destination. As the last task performed on a reduction in **codegen**, the destination is broken up by adjusting the register model appropriately, and replacing the right hand side by the non terminal "e".

All doubly qualified double registers must have a blank before the second "." to disambiguate them from floating point literals.

3.3. Code Generator Tables

The logical data structures for the code generator parser are described in the walk through **analysis**. These structures include the action, next and reduce tables. Auxiliary tables, also constructed by **analysis**, are reduce and default lists. **Analysis** uses matrices as the physical representation for these tables; **merge** breaks the matrices into shared slices, compresses the slices together contiguously, and constructs a list of pointers to slices for each row in the tables. These transformations and structures are described in the **merge** walkthrough.

4. Tmdl Tour

Tmdl uses standard compiler technology to map the TMDL source language onto data structures describing registers, symbols and rules. **Tmdl** was an uninteresting program to write, since it is basically a straight-forward syntax directed compiler using standard techniques. Hence, its general philosophy is quick, dirty and lazy, relying on available tools for some of the work. **Tmdl** does little syntactic error recovery (using panic mode) and no semantic error recovery, but does a good job of finding and reporting violations in the TMDL semantics.

Tmdl uses a scanner shared with **codegen**, ensuring common lexical conventions in the TMDL and the IR. The author originally wrote the scanner in *lex* (a lexical analyzer generator), but this 'tool' proved to be a waste of time. Regular expressions describing complicated patterns never worked reliably, the lexical description was hard to modify, development iteration time was long because *lex* is very slow, and execution time was intolerably slow for the code generator, because scanners constructed by *lex* are not efficient. The *lex* scanner was converted to a C scanner, which is easier to understand than the *lex* scanner.

Syntactic analysis of TMDL is driven by a *yacc* constructed parser. Since the *yacc* grammar for TMDL is small, *yacc* has none of the problems afflicting lex that were mentioned in the previous paragraph. In **tmdl**, semantic processing is syntax directed; semantic information is collected piecemeal and stuffed into static areas, and at appropriate times, gathered together, checked for semantic integrity, and stuffed into the structure needed for subsequent passes. As per convention, **tmdl** allocates most things dynamically, but imposes static restrictions on the number of grammar symbols (100), the number of registers (100) and the length of any production (100). Both lexical and syntactic analysis work together to allow most meta characters in TMDL to be declared and used as grammar symbols.

5. Analysis Algorithmic Tour

The tour through **analysis** will be conducted in two separate parts. The first part of the tour will re-present all the pertinent algorithms from Glanville's dissertation; this presentation is necessary to correct and extend some of the algorithms, and unify the theory behind the algorithms with the present implementation. The second part of the tour briefly describes the actual implementation, discussing how implementation decisions to save both space and time are reflected in modified versions of Glanville's algorithms, and conversely, how unobvious parts of the algorithms are reflected in the implementation. The second tour is short because much of the description is in the first tour.

5.1. Analysis Algorithms

5.1.1. Algorithm Merging

The crucial parts of the four major code generator-generator algorithms which Glanville presented sequentially in his dissertation, and implemented as sequential phases, have been merged so they are applied in (pseudo) parallel. These four algorithms are:

Number	*Title*
4.3	SLR(1) code generator construction
4.4	loop detection
4.5	loop elimination
4.6	uniformity testing

The fifth major algorithm, Algorithm 4.8 (default list construction) has not been merged, but is still done in the same physical phase.

Glanville presents his five major algorithms sequentially because it aids clarity, helps proceduralization, and keeps all the figures on one page. However, the algorithms in Glanville's dissertation are a direct reflection of the algorithms used in his implementation. That implementation was structured into four distinct phases, where each phase implemented one or two of the five major algorithms. Because Glanville's implementation split the equivalent of **analysis** across many phases, proper and general code for housekeeping, such as state addition or table lookup, was implemented in only one phase. Since default list construction and loop removal require these housekeeping functions, and since these housekeepers were in different phases, Algorithms 4.5 and 4.8 in Glanville's dissertation, directly reflecting the first implementation, contain pieces that do the housekeeping in a non general way.

All the merged algorithms possess a small subset of common characteristics. They iterate through states and require information only on the state shifted into from a given state. In addition, they iterate through all of the symbols, and require access to precomputed itemsets and relations. The four state iteration loops can be fused together, as well as some of the symbol iteration loops, provided that the fused loop bodies are sequentially applied in the proper order. The fifth algorithm, computing the default lists, cannot be fused together with the other four, since it must be applied after all loops are eliminated, and requires nearly random access to the information describing the states in the parser. However, constructing default lists does need many of the sets and support routines of the first four algorithms, so is included in the same physical phase as a logically separate phase.

5.1.2. Presentation Style

The following presentation of Glanville's algorithms is intended to reflect the same style as the originals, to fix some typographical and omission errors, explain the algorithms in more detail, and justify the loop fusion. The definitions, theory, proofs and high level descriptions in Glanville's dissertation, unless otherwise stated, still apply, and the reader should be thoroughly familiar with them. The algorithms are presented in the same descriptive set theoretic notation Glanville used, with many of the same variable names, or with names substituted into something more meaningful. The notation has been extended to include domain and range declarations of non obvious set and array variables. It is now well bracketed by keywords: (**begin**, **end**), ($\forall$ · · · **do**, **od**), (**while** · · · **do**, **od**), (**if** · · · **then**, **fi**), (**if** · · · **then** · · · **else**, **fi**), (**repeat**, **until**), and (**case**, **esac**), with PASCAL like semantics.

Each line in the algorithm has a marginal comment. These comments consist of a name and sequence number, and are used extensively in the accompanying text. The text describes the lower level decisions intended to support the theory in Glanville's dissertation, and provides comments lacking in the algorithms and in his dissertation. The text references all known errors in his dissertation by their corresponding (correct) line in these algorithms.

Fusing loops was one of the biggest changes to Glanville's algorithms. Since the implementation works with the kernel of states, instead of manipulating cores of states (see §5.1.3), the algorithms here are also changed to reflect this. In addition, global tests (also called *blanket* tests) asserting general existence of a condition are applied before any specific instances of that condition are searched for and found. For example, a blanket test determines that some symbol is **shifted**, and another test, in conjunction with an iterator, determines exactly which symbol is **shifted**.

This presentation is not without its flaws. The presented algorithm does not attempt to proceduralize for organization's sake, and does not proceduralize to avoid code replication. The main loop extends across many pages, and the gross structure can't be seen in one glance. (For clarity, some end brackets have comments to reflect this larger structure.) The marginal comments should aid in viewing the entire algorithm at a higher level of abstraction. The implementation is highly proceduralized.

5.1.3. declare.10 – declare.45 (Figure 5.1)

It is convenient to discuss here the domains, ranges and properties of the sets we manipulate.

Formally, the *machine description grammar* **G** is a four tuple, $\mathbf{G} = (\mathbf{N}, \Sigma, \mathbf{P}, \lambda)$.

N is the set of non terminals including λ. Recall that non terminals stand for registers.

Σ is the set of terminals.

```
declare I is a set of all items from syntactically distinct rules ∈ P;     declare.10
declare Q ∈ I* ordered by insertion order;                                 declare.15
declare R ∈ I* ordered by insertion order;                                 declare.20
declare ACTION: Q × V → (accept, shift, reduce, error) ;                  declare.25
declare NEXT: Q × V → Q;                                                  declare.30
declare DISTANCE: N × N → naturals;                                       declare.35
declare MINDISTANCE: N → naturals;                                        declare.40
declare CHAINLOOP ∈ N × N;                                                declare.45
```

Figure 5.1: Parser Construction Declarations

λ is the distinguished starting symbol, representing "no result". In alternative notations, λ is also written as "$", ".".

P is the set of productions describing the effect of each available instruction on the target machine. There may be more than one production for some target machine instructions. (We will sometimes loosely refer to P as the grammar.) Some productions may be syntactically identical (contrasted with syntactically distinct), differing only in semantic restrictions on the syntactic symbols. Two productions are syntactically distinct if the sequence of grammar symbols from $\Sigma \cup \mathbf{N}$ composing either the right or left hand sides of the productions differ; otherwise they are syntactically identical. In addition, there may be some productions that are syntactically identical, and to the extent specified in the TMDL, semantically identical.

V is the total vocabulary, $\mathbf{V} = \Sigma \cup \mathbf{N}$.

Semantically identical productions arise if the distinguishing nature of an idiomatic target machine instruction is not specified in the TMDL. As an example from the VAX,

r.1	→	+ r.1 k.2	**addl2**	k.2, r.1
r.1	→	+ r.1 k.2	**movab**	k.2(r.1), r.1
r.1	→	+ r.1 k.2	**movaw**	k.2(r.1), r.1
r.1	→	+ r.1 k.2	**moval**	k.2(r.1), r.1
r.1	→	+ r.1 k.2	**movad**	k.2(r.1), r.1

all do the same thing; there is one **mova** instruction that is costs less (is more efficient) than the others, depending on the semantic value of "k.2". Since this cost measure is not specified in the TMDL, all five instructions appear to the code generator to have the same cost, so the code generator could select any one of the five candidates, and still generate correct code. In the current implementation, the code generator selects the instruction whose production appears first in the machine description grammar.

We review here some definitions used in this report that are also defined (sometimes differently) in Glanville's dissertation. If "$v \rightarrow \alpha\, \beta$" is a grammar rule in P, where v is a non terminal from N, then $[v \rightarrow \alpha \cdot \beta]$ is an LR(0) *item*. If $|\alpha| = 0$, then the item is an *initial* item. If both $|\alpha| > 0$ and $|\beta| > 0$, then the item is a *medial* item. If $|\beta| = 0$, then the item is a *terminal* item. The rule "$v \rightarrow \alpha\, \beta$" is said to *source* the item $[v \rightarrow \alpha \cdot \beta]$. The rule sources $|\alpha| + |\beta| + 1$ items. An *itemset* is a set of items. A *sparse* itemset has only a few members of those possible. In a *kernel* itemset, all member items are either medial items, or they are initial items of the form $[\lambda \rightarrow \cdot \alpha]$. A kernel itemset that has been *closed* by augmenting with initial items as defined by function *close* (§5.3) is a *core* itemset. Since unique itemsets are associated one to one with *states* in the parser, we also speak of the kernel of a state and the core of a state. Note that the definitions of kernel and core used here follow those used in [AhoUllman77], and are different from the definitions in Glanville's dissertation. (A core in his dissertation is our kernel.)

I, in this implementation, is the set of all LR(0) items from syntactically distinct productions from **P**. By considering only syntactically distinct productions in a large grammar, the size of $|\mathbf{I}|$ was reduced by 14%, a worthwhile gain. Accounting for syntactically identical productions is more efficiently done when all set manipulations are completed.

Q is an ordered set of itemsets from **I**, one itemset for each state, ordered by insertion sequence. Only the kernel of a state is used as a member of **Q**, as the kernel uniquely identifies the core of the state. Since kernel itemsets are typically sparse, they can be efficiently represented as a list to conserve space. The only place a core is generated and manipulated is in the state expansion loop for the state currently being considered.

R is an ordered set of itemsets from **I**, ordered by insertion sequence. There is one itemset member of **R** for each unique reduction set that can be applied whenever the parser will **reduce**. All items in a given itemset element of **R** have terminal dots. Further, the items have syntactically identical right hand sides, and the left hand sides are all lambda, or are all weakly connected in the distance graph (see §5.1.4). Elements of **R** are eventually converted into sets of productions from **P**. The operations on the sets **R** and **Q** during parser construction are nearly identical.

5.1.4. distance.10 – distance.75 (Figure 5.2)

This code statically analyzes chain rule productions, and is taken directly from from Glanville's dissertation (Algorithm 4.5). The graph distance from u to v is the minimum cost to reduce non terminal u to non terminal v using only chain rules. Note that since $v \rightarrow u$, the arrow notation to express derivations in a sequence of productions is confusingly reversed from the graph connectivity expressed in the *distance* matrix. The algorithm uses *distance* to find the shortest sequences of chain reductions out of a looping configuration. We use a cost of 1 for all chain rules, as chain rules in all instruction grammars we have analyzed only have register allocation side effects, and do not produce code. Note the typo in Glanville's *distance.25*, where he has the order of the right and left sides of the production $v \rightarrow u$ interchanged. *Distance.40– distance.75* is Warshall's min closure of a digraph.

5.1.5. genstates.50 – genstates.75 (Figure 5.3)

The **while** loop in *genstates.50* iterates over all unconsidered states. An unconsidered state has a computed kernel, but the Action and Next values for that state are unknown. This **while** loop, in conjunction with the way unconsidered states are added to **Q**, traverses the parser's finite state machine control, viewed as a digraph,

```
procedure generatestates;                                                  genstates.10
begin                                                                      genstates.15
    declare $qkernel_j$, $qclosed_j \in$ I s.t. $1 \le j \le \infty$;      gendecl.10
    /* one $qkernel_j$ and $qclosed_j$ for each state */                   gendecl.15
    declare $qkernel' \in$ I;                                              gendecl.20
    declare $i \in$ I;                                                     gendecl.25
    declare $d, x \in$ N;                                                  gendecl.30
    declare $u, v, w \in$ V;                                               gendecl.35
    declare $curstate, maxstate, maxreduce \in$ naturals;                  gendecl.40

    $\forall u, v \in$ N do                                                distance.10
        DISTANCE[$u, v$] $\leftarrow \infty$;                              distance.15
    od                                                                     distance.20

    $\forall p \in$ P s.t. $p$ = "$v \rightarrow u$", $u \in$ N, $v \in$ N do     distance.25
        DISTANCE[$u, v$] $\leftarrow$ 1;                                   distance.30
    od                                                                     distance.35

    $\forall w \in$ N do                                                   distance.40
        $\forall u \in$ N do                                               distance.45
            $\forall v \in$ N do                                           distance.50
                DISTANCE[$u, v$] $\leftarrow$ min(DISTANCE[$u, v$],        distance.55
                        DISTANCE[$u, w$] + DISTANCE[$w, v$]);              distance.60
            od                                                             distance.65
        od                                                                 distance.70
    od                                                                     distance.75
```

Figure 5.2: State Generation Initialization

```
    $qkernel_0 \leftarrow \{i \in \mathbf{I} \mid i = [\lambda \rightarrow \cdot\alpha]\}$;          genstates.20
    $\mathbf{Q} \leftarrow qkernel_0$;                                         genstates.25
    $curstate \leftarrow 0$;                                          genstates.30
    $maxstate \leftarrow 0$;                                          genstates.35
    $maxreduce \leftarrow 0$;                                         genstates.40
    $\mathbf{R} \leftarrow \phi$;                                             genstates.45
    while $curstate \leq maxstate$ do                                   genstates.50
        CHAINLOOP $\leftarrow \phi$;                                       genstates.55
        $qclosed_{curstate} \leftarrow close(qkernel_{curstate})$;             genstates.60

        $\forall v \in \mathbf{V}$ do                                            genstates.65
            ACTION[$qkernel_{curstate}$, $v$] $\leftarrow$ error;                genstates.70
        od                                                          genstates.75

        Fig. 5.4:"Construction of $shift$ Actions"                  seefig.10
        Fig. 5.5:"Construction of $reduce$ Actions"                 seefig.15
        Fig. 5.6:"Construction of the Minimum Distance              seefig.20
                  out of a Looping Configuration"                   seefig.25
        Fig. 5.8:"Loop Removal"                                     seefig.30
        Fig. 5.9:"Blocking Detection"                               seefig.35

        $curstate \leftarrow curstate + 1$;                                 genstates.80
    od /* uncompleted state iterator */                             genstates.85
    ACTION[$qkernel_0$, \$] $\leftarrow$ accept;                                genstates.90
end generatestates;                                                 genstates.95
```

Figure 5.3: Unconsidered State Iterator (Top Level)

in breadth first order. *Genstates.55* is from Algorithm 4.4 of Glanville's dissertation, and initializes detection of looping configurations in this state. See the unnumbered figure in Glanville's dissertation on page 68. Next, we close the kernel itemset of the current state, and consider this core only within the body of the state iterator. Initially, all actions are **error**.

5.1.6. shift.10 – shift.75 (Figure 5.4)

Testing for **shifts** is done first to force a **shift** if there is a **shift/reduce** conflict. This algorithm is directly from Glanville Algorithm 4.3, except for the global test in *shift.10*. *Shift.40 – shift.75* enters the kernel itemset for the new state into **Q**, extending **Q** as necessary.

5.1.7. loopdetect.10 – loopdetect.50 (Figure 5.4)

This code is the body of Algorithm 4.4 from Glanville's dissertation. At this point, the kernel for the state shifted into from this state has been constructed. We need this kernel to find chain reductions causing a looping configuration. *Chainloop*

```
/* see if some symbols shifts */                                        comment.10
if $\exists\ i \in qclosed_{curstate},\ v \in \mathbf{V}$ s.t. $i = [x \to \alpha \cdot v\ \beta]$ then     shift.10
    /* iterate through all symbols $v$ */                               comment.15
    $\forall\ v \in \mathbf{V}$ do                                      shift.15
        /* see if $v$ shifts */                                         comment.20
        if $\exists\ i \in qclosed_{curstate}$ s.t. $i = [x \to \alpha \cdot v\ \beta]$ then     shift.20
            ACTION[$qkernel_{curstate},\ v$] $\leftarrow$ shift;        shift.25
            $qkernel' \leftarrow \{[x \to \alpha\ v \cdot \beta] \in \mathbf{I} \mid [x \to \alpha \cdot v\ \beta]$     shift.30
                    $\in qclosed_{curstate}\}$;                         shift.35

            /* see if $qkernel_j$ is already entered */                 comment.25
            if $\exists\ qkernel_j \in \mathbf{Q}$ s.t. $qkernel_j = qkernel'$ then     shift.40
                NEXT[$qkernel_{curstate},\ v$] $\leftarrow qkernel_j$;  shift.45
            else                                                        shift.50
                $maxstate \leftarrow maxstate + 1$;                     shift.55
                $qkernel_{maxstate} \leftarrow qkernel'$;               shift.60
                NEXT[$qkernel_{curstate},\ v$] $\leftarrow qkernel_{maxstate}$;     shift.65
                $\mathbf{Q} \leftarrow \mathbf{Q} + qkernel'$;          shift.70
            fi /* $qkernel_j$ is not entered */                         shift.75

            if $v \in \mathbf{N}$ then                                  loopdetect.10
                if $\exists\ i \in qkernel'$ s.t. $i = [x \to d \cdot]$,     loopdetect.15
                        $x \in \mathbf{N},\ d \in \mathbf{N}$ then      loopdetect.20
                    $\forall\ i \in qkernel'$ s.t. $i = [x \to d \cdot]$,     loopdetect.25
                            $x \in \mathbf{N},\ d \in \mathbf{N}$ do    loopdetect.30
                        CHAINLOOP $\leftarrow$ CHAINLOOP + $\{ (d,\ x) \}$;     loopdetect.35
                    od                                                  loopdetect.40
                fi                                                      loopdetect.45
            fi /* shift on $v \in \mathbf{N}$ */                        loopdetect.50

        fi /* shift on $v$ possible */                                  shift.80
    od /* symbol iterator */                                            shift.85
fi /* some symbol shifts */                                             shift.90
```

Figure 5.4: Construction of shift Actions

has the same graph connectivity conventions as *distance*: $(d,\ x)$ is in *chainloop* if $x \to d$, or d reduces to x.

5.1.8. reduce.10 – reduce.105 (Figure 5.5)

This code is directly from Algorithm 4.3 in Glanville's dissertation, although there is again a global test to ascertain a potential reduction before testing each symbol individually. (We assume that the relation FOLLOW has already been computed using the definition from his dissertation on page 75.) *Reduce.45* ensures that **reduce/reduce** conflicts are resolved in favor of the rule with the longest right hand side; the left hand side is not taken into consideration. *Reduce.55* through *reduce.95* enter the itemset of terminal items for the new reduce set into **R**, much

```
if $\exists\ i \in qclosed_{curstate}$ s.t. $i = [x \rightarrow \alpha \cdot]$ then                reduce.10
   $\forall\ v \in$ V do                                                                     reduce.15
      if ACTION[$curstate$, $v$] = error, $v \in$ FOLLOW($x$) then                           reduce.20
         ACTION[$qkernel_{curstate}$, $v$] $\leftarrow$ reduce;                              reduce.25
         $r' \leftarrow \{[x \rightarrow \alpha \cdot] \in qclosed_{curstate} \in$ I |       reduce.30
               $v \in$ FOLLOW ($x$) and                                                      reduce.35
               $\nexists\ i \in qclosed_{curstate}$ s.t. $i = [x' \rightarrow \alpha' \cdot]$,   reduce.40
               $length(\alpha') > length(\alpha)$,                                           reduce.45
               $v \in$ FOLLOW($x'$)};                                                        reduce.50
         if $\exists\ r_j \in$ R s.t. $r_j = r'$ then                                        reduce.55
            NEXT[$qkernel_{curstate}$, $v$] $\leftarrow r_j$;                                reduce.60
         else                                                                                reduce.65
            $maxreduce \leftarrow maxreduce + 1$;                                            reduce.70
            $r_{maxreduce} \leftarrow r'$;                                                   reduce.75
            NEXT[$qkernel_{curstate}$, $v$] $\leftarrow r_{maxreduce}$;                      reduce.80
            R $\leftarrow$ R + $r'$;                                                         reduce.85
         fi                                                                                  reduce.90
      fi /* $v$ reduces */                                                                   reduce.95
   od /* symbol iterator */                                                                  reduce.100
fi /* some symbol reduces */                                                                 reduce.105
```

Figure 5.5: Construction of reduce Actions

as was done for Q in *shift.40 – shift.70*.

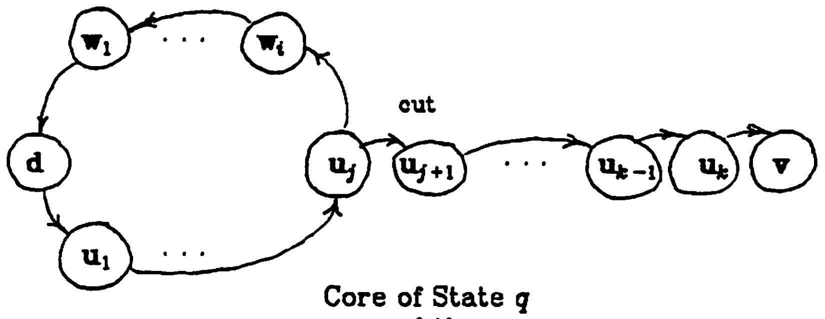
```
/* closing fi  for looprm.10 is looprm.155, fig 5.8 */                                       comment.30
if $\exists\ d \in$ N s.t. $(d, d) \in$ CHAINLOOP$^+$ then                                   looprm.10

   $\forall\ v \in$ N do                                                                     mindist.10
      MINDISTANCE[$v$] $\leftarrow \infty$;                                                  mindist.15
   od                                                                                        mindist.20
   $\forall\ v \in$ N s.t. $[y \rightarrow \alpha \cdot v\ \beta] \in qclosed_{curstate}$, $\alpha \neq \lambda$ do   mindist.25
      MINDISTANCE[$v$] $\leftarrow$ 0;                                                       mindist.30
      $\forall\ x \in$ N do                                                                  mindist.35
            MINDISTANCE[$x$] $\leftarrow$ min(MINDISTANCE[$x$],                              mindist.40
                  DISTANCE[$x$, $v$]);                                                       mindist.45
      od                                                                                     mindist.50
   od                                                                                        mindist.55
```

Figure 5.6: Construction of the Minimum Distance out of a Looping Configuration

5.1.9. looprm.10 – mindistance.50 (Figure 5.6)

This code is directly from Algorithm 4.5 in Glanville's dissertation. At this point we have constructed the kernels of all states shifted into from this state, and have enough information to remove any looping configuration by splitting states. *Looprm.10* checks if there is a chain rule reduction loop, using the information gathered in *shift*. Figure 5.7 schematically shows part of the complete itemset for state q, together with the copied, complete and documented graph from Glanville's dissertation Figure 4.5. The graph is represented as a cost matrix, as Glanville suggests. For each symbol, *mindistance* is the cost of reduction using chain rules out of a loop to the closest "safety" symbol (x in *mindistance.25*) that shifts out of the looping configuration.

state q: $[x \rightarrow \alpha \cdot v\beta]$ *{part of the core}*
$[v \rightarrow \cdot u_k]$
$[u_k \rightarrow \cdot u_{k-1}]$
. . .
$[u_{j+1} \rightarrow \cdot u_j]$
$[u_j \rightarrow \cdot u_{j-1}]$
. . .
$[u_2 \rightarrow \cdot u_1]$
$[u_1 \rightarrow \cdot d]$
$[d \rightarrow \cdot w_1]$
$[w_1 \rightarrow \cdot w_2]$
. . .
$[w_i \rightarrow \cdot u_j]$ *{removed}*



Core of State q
and the
Graph Constructed to Eliminate the Loop
$\mathbf{d} \rightarrow \cdots \rightarrow \mathbf{d}$
(From Figure 4.5 of Glanville's Dissertation)

Figure 5.7: Loop Removal Graph

```
  /* iterate through all non terminals */                                      comment.35
  $\forall$ $v \in$ N do                                                       looprm.15
     /* determine if $v$ shifts */                                             comment.40
     if ACTION[$qkernel_{curstate}$, $v$] = shift,                             looprm.20
           $\exists$ $i \in qclosed_{curstate}$ s.t. $i = [y \rightarrow \alpha \cdot v\ \beta]$,   looprm.25
           $\alpha \neq \lambda$ then                                          looprm.30
        $qkernel' \leftarrow \phi$;                                            looprm.35
        $dropped \leftarrow$ false;                                            looprm.40
        $qkernel_j \leftarrow$ NEXT[$qkernel_{curstate}$, $v$];                looprm.45
        $\forall$ $i \in qkernel_j$ s.t. $i = [v' \rightarrow v\ \cdot]$ do    looprm.50
           if MINDISTANCE[$v'$] < MINDISTANCE[$v$] then                        looprm.55
              $qkernel' \leftarrow qkernel' + i$;                              looprm.60
           else                                                                looprm.65
              $dropped \leftarrow$ true;                                       looprm.70
           fi                                                                  looprm.75
        od                                                                     looprm.80

        /* if we drop an item, then we must split the state */                 comment.45
        if $dropped$ then                                                      looprm.85
           assert $qkernel' \neq \phi$;                                        looprm.90
           assert $close(qkernel') = qkernel'$;                                looprm.95
           if $\exists$ $qkernel_j \in$ Q s.t. $qkernel_j = qkernel'$ then     looprm.100
              NEXT[$qkernel_{curstate}$, $v$] $\leftarrow qkernel_j$;          looprm.105
           else                                                                looprm.110
              $maxstate \leftarrow maxstate + 1$;                              looprm.115
              $qkernel_{maxstate} \leftarrow qkernel'$;                        looprm.120
              NEXT[$qkernel_{curstate}$, $v$] $\leftarrow qkernel_{maxstate}$; looprm.125
              Q $\leftarrow$ Q + $qkernel'$;                                   looprm.130
           fi                                                                  looprm.135
        fi /* split a state */                                                 looprm.140
     fi /* shift on $v$ */                                                     looprm.145
  od /* symbol iterator */                                                     looprm.150

fi /* a chain loop is possible (tied to looprm.10) */                          looprm.155
```

Figure 5.8: Loop Removal

5.1.10. looprm.15 – looprm.155 (Figure 5.8)

This code is based on the latter half of Glanville algorithm 4.5, with some important differences correcting omissions in that algorithm. First, the action from the current state on a given non terminal must **shift** (*looprm.20*). Secondly, there is no point in splitting the state we **shift** into if that state gets us out of the looping configuration (*looprm.25*). Thirdly, we only add a new state to **Q** if a split state is created and is unique from all others in **Q** (*looprm.100 – looprm.140*). (Glanville's myopic addition of states is a vestige of his implementation.) The **if** statement in *looprm.55* checks if the chain reduction in the state shifted into is indeed along the shortest reduction path out of the loop. If it is not the shortest path, that chain reduction will be dropped from the reduction set of a new state added to replace state j (the split state). The arc in the loop corresponding to the dropped chain reduction (the arc u_j to w_i) is where the graph is "cut" to break the loop. Because of the calculations done in *mindistance.15* – *mindistance.60*, there are one or more chain reductions that will eventually get considered by the non terminal iterator in *looprm.20*, one of which is the shortest path out of the loop. This observation can be used to prove the assertion in *looprm.90*. The assertion in *looprm.95* follows directly from the properties of terminal items in reduce itemsets. The construction of reduce actions for the new state will eventually get done when the state iterator considers this new unconsidered state, not here as Glanville used.

5.1.11. block.10 – block.85 (Figure 5.9)

This is the body of Algorithm 4.6 from Glanville's dissertation. The algorithm can be applied now, as actions for all symbols in this state have been computed.

5.1.12. close.10 – close.45 (Figure 5.10)

This function is taken verbatim from Algorithm 4.3 in Glanville's dissertation.

5.1.13. Reduce and Default List Construction

Reduce and default lists are discussed by Glanville in his dissertation, §4.5.7 on pages 82-84. However, Algorithm 4.8 contains serious omissions, so some of the discussion is repeated below with this in mind.

```
∀ $i \in qclosed_{curstate}$ s.t. $i = [r \rightarrow \alpha \cdot v\ \beta]$, $\alpha \neq \lambda$, $v \in \mathbf{V}$ do      block.10
   $x \leftarrow parent(v, "r \rightarrow \alpha v \beta")$;                                          block.15
   if $leftchild(v, "r \rightarrow \alpha v \beta")$ then                                       block.20
      ∀ $u$ s.t. $x$ LEFT FIRST $u$ do                                                 block.25
         if ACTION[$qkernel_{curstate}$, $u$] = error then                                 block.30
            output ("Not Uniform.");                                                   block.35
         fi                                                                            block.40
      od                                                                               block.45
   else                                                                                block.50
      ∀ $u$ s.t. $x$ RIGHT FIRST $u$ do                                                block.55
         if ACTION[$qkernel_{curstate}$, $u$] = error then                                 block.60
            output ("Not Uniform.");                                                   block.65
         fi                                                                            block.70
      od                                                                               block.75
   fi /* testing right child */                                                        block.80
od /* itemset iterator to test for blocks */                                           block.85
```

Figure 5.9: Blocking Detection

```
function close ( q );
   declare $q \in I$;
begin
   repeat
      $q \leftarrow q + \{[x \rightarrow \cdot\alpha] \in I \mid [y \rightarrow \beta \cdot x\ \gamma] \in q, x \neq \lambda\}$;
   until q does not change;
   return ( q );
end close;
```

Figure 5.10: Function Close

5.1.13.1. Reduce Lists

When the parser constructed in the previous section performs a reduce action using a reduce itemset $r \in \mathbf{R}$, it is prepared to reduce by one or more productions from **P** having right hand sides with a particular syntax. The reduce itemset constructor has already disregarded syntactically distinct productions that are shorter (and presumably not as efficient) as other potential productions, so all items in the reduce itemset have identical right hand sides. However, as will be recalled, the domain of **R** includes only syntactically distinct productions from **P**. The reduce list (or *rlist*) constructor converts all reduce itemsets in **R** into lists of production numbers from the entire grammar **P**. (See Figure 5.11.) Rlists are sorted on the major key of descending semantic restriction level, with a minor key of production enumeration order in the TMDL. **Codegen** will select the first rule from the reduce list whose semantic attributes match the semantic attributes of the isolated pattern (see §8.4). Regardless of the left hand side, any rule from an rlist matching the major key could be chosen, since by adherence to the uniformity conditions, the left hand side will always be **shifted** after the reduction. However, the choice of left hand side may seriously affect the overall code efficiency. Unfortunately, as discussed in §8.9, **codegen** is only a local decision maker. Ordering by minor key allows the TMDL author to exercise some control in lieu of the cost function suggested on the bottom of page 83 of Glanville's dissertation.

5.1.13.2. Semantic Blocking and Default Lists

If no rule on the applied rlist semantically matches the pattern, **codegen** is said to *semantically block*. **Analysis** anticipates this block, and for each fully restricted reduce list, constructs a default list (or *dlist*) that **codegen** recursively applies to the pattern. **Analysis** builds the default lists by constructing a small code generator for a subsetted target machine. It then emits code using the subsetted code generator for the right hand side of any restricted rule (the *prototype*) taken from the reduce list. This emitted code, encoded as a sequence of reductions, their application order and their position, is encapsulated into a default list associated with the reduce list. Applying the dlist at **codegen** time produces code composed from simpler instructions from the subsetted machine; these instructions implement the more complicated, semantically restricted rule. For an example, see §8.5.

```
procedure expandR;                                                              eR_decl.10
begin                                                                           eR_decl.15
   declare $r \in R$ ordered by insertion order;                                eR_decl.20
   declare $prodset \in P$ ordered by semantic restriction, high to low;        eR_decl.25
   declare $default \in (I, naturals)^*$ ordered by insertion order;            eR_decl.30
   declare $i \in I$;                                                           eR_decl.35
   declare $g \in P$;                                                           eR_decl.40
   declare $restricted \in$ boolean;                                            eR_decl.45

   $\forall r \in R$ do                                                         eR_rist.10
      $prodset \leftarrow \phi$;                                                eR_rest.10
      $default \leftarrow \phi$;                                                eR_rest.15

      $restricted \leftarrow$ true;                                             eR_rest.20
      $\forall i \in r$ s.t. $i = [x' \rightarrow \alpha' \bullet]$, $x' \in N$ do   eR_rest.25
         $\forall$ instructions $g \in P$, $g = "x' \rightarrow \alpha'"$ do    eR_rest.30
            $prodset \leftarrow prodset + g$;                                   eR_rest.35
            $restricted \leftarrow restricted \wedge (g$ is restricted$)$;      eR_rest.40
         od                                                                     eR_rest.45
      od                                                                        eR_rest.50

      if restricted then                                                        eR_default.10
         $default \leftarrow dlist(r)$;                                         eR_default.15
      fi                                                                        eR_default.20
      $r \leftarrow$ map($prodset$, $default$);                                 eR.10
   od /* of iterating through all $r \in R$ */                                  eR.15
end /* expandR */                                                               eR.20
```

Figure 5.11: Expansion of Reduce Lists

5.1.13.3. Grammar Subsetting and Skeleton Parser Construction

The code generator for the subsetted target machine is constructed on-the-fly during the parse of the prototype. (Refer to figure 5.12.) The target machine grammar is subsetted to contain only productions with right hand sides of length less than or equal to the length of the prototype, excluding rules with the same right hand side as the prototype, as done in *dl_lgrest.20*. An itemset containing items sourced by these restricted rules is said to be length restricted. The length restriction should be contrasted with the length restriction Glanville used in his dissertation (page 83), where rules with length strictly less than the prototype are chosen, and in his dissertation Algorithm 4.8, where rules with length less than or equal to the prototype are chosen. The first restriction may produce inefficient code, the second produces degenerate default lists; both are wrong.

During default list construction, both the itemsets used in the unrestricted parser construction and the action and next tables of the unrestricted parser (the *skeleton* parser) are readily available. The prototype is parsed with the skeleton

```
function $dlist(r)returns(\mathrm{I},\ \mathbf{naturals})^*$;                                        dl.10
    declare $r \in \mathbf{R}$;                                                                    dl.15
begin                                                                                              dl.20
    declare $default \in (\mathrm{I},\ \mathbf{naturals})^*$ordered by insertion order;            dl_decl.10
    declare $r,\ r_{short} \in \mathbf{R}$ unordered;                                              dl_decl.15
    declare $lgrestricted \in \mathrm{I}$;                                                         dl_decl.20
    declare $qstart \in \mathrm{I}$;                                                               dl_decl.25
    declare $q,\ q_{short} \in \mathrm{I}$;                                                        dl_decl.30
    declare $v \in \mathbf{V}$;                                                                    dl_decl.35
    declare $q',\ q'_{short} \in \mathrm{I}$;                                                      dl_decl.40
    declare $v' \in \mathbf{V}$;                                                                   dl_decl.45
    declare $readhead,\ j,\ k \in$ naturals;                                                       dl_decl.50
    declare $i \in \mathrm{I}$;                                                                    dl_decl.55
    declare $\rho \in (\mathbf{V} \cup \lambda)^*$;                                                dl_decl.60
    declare $x,\ y,\ z \in \mathbf{N}$;                                                            dl_decl.65
    declare $goals,\ explored,\ toexplore,\ nextexplore \in \mathbf{N}^*$;                         dl_decl.70

    $default \leftarrow \phi$;                                                                     dl_start.10
    /* find starting state */                                                                      dl_start.15
    Let $[z \rightarrow \alpha \cdot]$ be any $i \in r$;                                           dl_start.20
    if $z = \lambda$ then                                                                          dl_start.25
        $qstart \leftarrow qkernel_0$;                                                             dl_start.30
    else                                                                                           dl_start.35
        $qstart \leftarrow qkernel_k$ s.t. $\exists\ i \in qkernel_k$ s.t.                         dl_start.40
                $i = [y \rightarrow \beta \cdot z\ \gamma]$ and $\nexists\ j$ s.t. $j < k,\ i \in qkernel_j$;   dl_start.45
    fi                                                                                             dl_start.50
    assert $qstart$ is a state , $i \in close(qstart),\ i = [z \rightarrow \cdot\,\alpha] \in r$; dl_start.55
    $q \leftarrow qstart$;                                                                         dl_start.60

    $goals \leftarrow \phi$;                                                                       dl_goals.10
    $\forall\ i \in r,\ i = [z \rightarrow \alpha \cdot]$ do                                       dl_goals.15
        $goals \leftarrow goals + \{z\}$;                                                          dl_goals.20
    od                                                                                             dl_goals.25

    $lgrestricted \leftarrow \{[v \rightarrow \beta \cdot \gamma] \in \mathrm{I} \mid v \in \mathbf{N},\ |\beta\gamma| \leq |\alpha|\}$;   dl_lgrest.10
    $\forall\ j$ s.t. $0 \leq j \leq |\alpha|$ do                                                  dl_lgrest.15
        $lgrestricted \leftarrow lgrestricted - \{[x' \rightarrow \beta \cdot \gamma] \in \mathrm{I} \mid \alpha = \beta\gamma,\ |\beta| = j\}$;   dl_lgrest.20
    od                                                                                             dl_lgrest.25
    $readhead \leftarrow 1$;                                                                       dl_pset.10
    $\rho \leftarrow$ concatenate$(\alpha,\ \$^{|\alpha|})$;                                       dl_pset.15
    $push(q)$;                                                                                     dl_pset.20
```

Figure 5.12: Procedure Dlist (Set Up)

parser, checking in each state that the length restricted closed kernel itemset (the length restricted core) allows the action to be performed. Since the input string being parsed is known exactly, the restricted parser constructor need only construct a small fraction of the entire parser, and can use much of the existing skeleton. Consequently, some inefficiencies are acceptable. The two algorithms are different enough to warrant a separate implementation.

If the right hand side of the prototype rule is α, $\alpha = \alpha_1 \cdots \alpha_n$, then the restricted parser is started in the first state in the unrestricted parser that shifts on α_1 (*dl_start.55*). The restricted parser will finish parsing the prototype when it performs an **accept** action (only if the prototype's left hand side is λ), or when the prototype is completely **reduced** to a non terminal *goal* symbol that is shifted in the established start state, or can be shifted when the reduce list is applied (*dl_goals.10-dl_goals.25, dl_done.10-dl_done.25*). These are the possible intermediate actions, determined by examining length restricted itemsets and the skeleton. (See figure 5.13 for the implementing code.)

error The restricted machine (and undoubtedly the unrestricted machine) is non uniform, and **codegen** may still semantically block.

shift The symbol shifted will be contained in the right hand side of a length restricted production. The unrestricted skeleton parser's action and next tables are valid.

reduce A length restricted production is to be applied. A reduce list is constructed in the usual fashion from the length restricted reduce itemset (which is added to **R**, possibly to have an associated default list, implying a recursive application of reduce and default lists at **codegen** time). However, unlike an reduce list constructed for the **codegen** parser to use when not applying a default list, all left hand sides for productions on these reduce lists must be the same, so that the production dynamically chosen at **codegen** time has the same left hand side statically chosen by **analysis** during default list construction. (Otherwise, inconsistencies in **codegen**'s parse stack may cause **codegen** to loop applying default lists.) In addition, in the event of alternative left hand sides, the left hand side chosen must keep the restricted code generator away from looping configurations.

The skeleton parser was constructed so that it could not loop by repeated application of chain rules. However, since the dynamically constructed subset parser deviates from the skeleton, the subset parse may encounter looping configurations that could not occur (and would have been detected) in the skeleton parser. Consequently, the ideas from the loop detection and removal algorithms, §5.1.7 and §5.1.10 must be integrated into the subsetted parser. For example, a reduce list constructed from the PDP11 grammar consists of the single production:

```
r.1 → * k=2 r.1    "asl r.1";
```

This reduce lists requires a default list, because of the restriction on the constant. During default list construction, the prototype is parsed with this stack and input: (The tuples are (state q, symbol **shifted** while in state q))

stack	*input*
(1 *) (8 k) (18	r $ $ $

r can not be shifted because of length restriction. Hence, "$r \rightarrow k$" is applied, yielding

```
while $readhead \le sizeof(\rho)$ do                                                    dl_parse.10
   if $sizeof(\rho) = 1$ and $readhead = 1$ and $\rho_1 \in N$                          dl_done.10
         and ($\rho_1 \in goals$ or ACTION[$qstart, \rho_1$] = shift) then              dl_done.15
      break from enclosing while ;                                                      dl_done.20
   fi                                                                                   dl_done.25

   $v \leftarrow \rho_{readhead}$;                                                      dl_look.10
   $v' \leftarrow \rho_{readhead+1}$;                                                   dl_look.15
   case ACTION[$q, v$] of                                                               dl_parse.15
   accept:                                                                              dl_parse.20
      break from enclosing while ;                                                      dl_parse.25
   error:                                                                               dl_parse.30
do_error:                                                                               dl_error.10
      output ("No code for "$x \rightarrow \alpha$'"");                                 dl_error.15
      break from enclosing case ;                                                       dl_error.20

   shift:                                                                               dl_parse.35
      $q_{short} \leftarrow close(q \cap lgrestricted)$;                                dl_shift.10
      if $q_{short} = \phi$ then                                                        dl_shift.15
         break from enclosing while ;                                                   dl_shift.20
      fi                                                                                dl_shift.25

      if $\nexists\, i \in q_{short}$ s.t. $i = [x \rightarrow \alpha \cdot v\beta]$ then   dl_shift.30
         $q_{short} \leftarrow q_{short} \cap \{[y \rightarrow \gamma \cdot] \mid y \in N\}$;  dl_shift.35
         goto do_reduce;                                                                dl_shift.40
      fi                                                                                dl_shift.45
      if $v \in N$ then/* chain loops possible */                                       dl_shift.50

         Fig. 5.14:"Dlist Shift Action and Chain Reduction Detection"                   seefig.40

      fi /* chain loops possible */                                                     dl_shift.55
      $q \leftarrow$ NEXT[$q, v$];                                                      dl_shift.60
      push($q$);                                                                        dl_shift.65
      break from enclosing case ;                                                       dl_shift.70

   reduce:                                                                              dl_parse.40

      Fig. 5.15"Default List Reduce Action and Deletion of Chain Rules"                 seefig.45

      break from enclosing case ;                                                       dl_parse.45

   esac                                                                                 dl_parse.50
od /* symbols in $\rho$ iterator */                                                     dl_parse.55

return (default);                                                                       dl.25
end / *dlist* /                                                                         dl.30
```

Figure 5.13: Default List Construction Parser Simulator

stack	*input*
(1 *) (8 r) (19	r $ $ $

In the subsetted machine, since FIRST (r) = **V**, ACTION[19, $*$] is **shift**, so the loop detection and removal mechanism is not brought to bear in states 8, 18 and 19, precisely where we need it now. In this configuration, both $e.1 \rightarrow r.1$ and $o.1 \rightarrow r.1$ might be used. If $e.1 \rightarrow r.1$ is chosen, the next production is $r.1 \rightarrow e.1$ which loops; otherwise, the production $o.1 \rightarrow * \; o.1 \; r.1$ is chosen.

Because the input to the subsetted parser is known, and because a wealth of precomputed information exists about possible reductions, the full loop detection and removal algorithms need not be applied. Figure 5.14 is the code implementing the partial loop detection algorithms. For the general configuration:

stack	*input*
· · · (q v) (q'	v', $v \in$ **N**

the worst is assumed: v might, after repeated reductions, reduce in a loop to itself. This assumption yields unnecessary work, at the gain of simplicity. Unlike §5.1.9, CHAINLOOP is not filled and closed to see if there really is a loop. However, MINDISTANCE is used in an identical way: MINDISTANCE is the number of chain reductions that will be applied before the parser is out of a looping configuration. When in state q, MINDISTANCE is computed for possible future use in state q'. If q' **shifts** on v', then MINDISTANCE[v] = 0, as the **shift** decreases the size of the input. if q' **reduces** by a non chain rule on v', then MINDISTANCE[v] = 0, as the reduction decreases the depth of the stack. Otherwise, $z \rightarrow v$ and the reduction chain potentially forming a loop needs to be explored from z. MINDISTANCE is closed using the DISTANCE matrix computed for the unsubsetted machine. When in state q', with only chain reductions possible, MINDISTANCE is guaranteed to be valid (figure 5.15, *dl_reduce.50*). The chain reduction(s) reducing v to a non terminal that gets q closer to an escape loop is chosen, and reflected into the default list constructed by the parse of the prototype.

The default list construction differs from Algorithm 4.8 in Glanville's dissertation (page 84), not only with the definition of length restriction and loop avoidance techniques, but in other ways. Glanville's dissertation uses core itemsets; **analysis** uses kernels throughout. Instead of using an additional loop with the auxiliary variable *readflag* to control which symbols are used from the prototype, the default list constructor rewrites and collapses the prototype after each reduction, and backs up the *readhead* (*dl_reduce.180* – *dl_reduce.200*). This closely resembles default list application performed in **codegen**, and is conceptually simpler.

Appendix 1 shows that the default list construction phase of **analysis** accounts for up to 50% its execution time. In light of this, the default list construction algorithm may have to be rewritten.

```
$\forall$ $x \in$ N do                                                        dl_shift.75
    MINDISTANCE[$x$] $\leftarrow \infty$;                                     dl_shift.80
od                                                                            dl_shift.85
$explored \leftarrow \phi$;                                                   dl_shift.90
$toexplore \leftarrow \{v\}$;                                                 dl_shift.95

/* explore the lhs's of all chain rules */                                    comment.50
while $toexplore \neq \phi$ do                                                dl_shift.100
    $nextexplore \leftarrow \phi$;                                            dl_shift.105

    /* explore the non terminal $x$ */                                        comment.55
    $\forall$ $x$ s.t. $x \in toexplore$                                      dl_shift.110
            and $\exists\ i \in q_{short}$ s.t. $[y \rightarrow \alpha \cdot x\beta]$ do    dl_shift.115
        $q' \leftarrow$ NEXT[$q$, $x$];                                       dl_shift.120
        $q'_{short} \leftarrow close(q' \cap lgrestricted)$;                  dl_shift.125
        if $q'_{short} = \phi$ then                                           dl_shift.130
            break from enclosing $\forall$ ;                                  dl_shift.135
        fi                                                                    dl_shift.140
        if $v' \neq \lambda$ and $\exists\ i \in q'_{short}$ s.t. $= [y \rightarrow \alpha\ x \cdot v'\ \beta]$ then    dl_shift.145
            MINDISTANCE[$x$] $\leftarrow$ 0;                                  dl_shift.150
        else                                                                  dl_shift.155
            /* do a reduction */                                              comment.60
            $q'_{short} \leftarrow q'_{short} \cap \{[y \rightarrow \gamma \cdot]\}$;    dl_shift.160
            if $q'_{short} = \phi$ then                                       dl_shift.165
                break from enclosing $\forall$ ;                              dl_shift.170
            fi                                                                dl_shift.175

            /* Is this production not a chain loop? */                        comment.65
            if $\exists\ i \in q'_{short}$ s.t. $i = [y \rightarrow \delta\ z \cdot]$,    dl_shift.180
                    $|\delta| \geq 1$, $z \in$ V then                         dl_shift.185
                MINDISTANCE[$x$] $\leftarrow$ 0;                              dl_shift.190
            else/* chain reduction only */                                    dl_shift.195
                $\forall$ $[y \rightarrow x \cdot] \in q'_{short}$ do         dl_shift.200
                    if $y \notin explored$ then                               dl_shift.205
                        $nextexplore \leftarrow nextexplore \cup \{y\}$;      dl_shift.210
                    fi                                                        dl_shift.215
                od                                                            dl_shift.220
            fi /* of just a chain reduction */                                dl_shift.225

        fi /* of doing a reduction */                                         dl_shift.230
        $explored \leftarrow explored \cup \{x\}$;                            dl_shift.235

    od /* exploring a non terminal */                                         dl_shift.240
    $toexplore \leftarrow nextexplore$;                                       dl_shift.245
od /* exploring all chain reductions */                                       dl_shift.250

/* minimize distances for explored non terminals */                           comment.70
$\forall$ $x \in$ N do                                                        dl_shift.255
```

```
    if $x \in explored$ then                                  dl_shift.260
        MINDISTANCE[$x$] ←                                    dl_shift.265
                min( MINDISTANCE[$x$], DISTANCE[$v$, $x$]);   dl_shift.270
    fi                                                        dl_shift.275
od                                                            dl_shift.280

assert $\exists$ $x$ s.t. MINDISTANCE[$x$]$\neq\infty$;       dl_shift.285
```

Figure 5.14: Dlist Shift Action and Chain Reduction Detection

```
$q_{short} \leftarrow close(q \cap lgrestricted)$;                                dl_reduce.10
if $q_{short} = \phi$ then                                                         dl_reduce.15
    break from enclosing while ;                                                   dl_reduce.20
fi                                                                                 dl_reduce.25
do_reduce:                                                                         dl_reduce.30
if $\nexists\, i \in q_{short},\, i = [y \rightarrow \delta\, z\, \cdot]\ |\delta| \geq 1$ then   dl_reduce.35
    /* we reduce via chain rule */                                                 dl_reduce.40

    /* Now, discard looping chain rules */                                         comment.75
    $\forall\, i \in q_{short}$ s.t. $i = [y \rightarrow z\, \cdot]$ do           dl_reduce.45
        if MINDISTANCE$[y] \geq$ MINDISTANCE$[z]$ then                            dl_reduce.50
            $q_{short} \leftarrow q_{short} - \{i\}$;                              dl_reduce.55
        fi                                                                         dl_reduce.60
    od                                                                             dl_reduce.65
    assert $(q_{short} \neq \phi)$;                                                dl_reduce.70
fi                                                                                 dl_reduce.75

let $[x_0 \rightarrow \alpha_0 \cdot] \in r_{short}$;                              dl_reduce.80
$\forall\, i \in r_{short},\, i = [x \rightarrow \alpha \cdot]$ do                dl_reduce.85
    if $x \neq x_0$ then                                                           dl_reduce.90
        $r_{short} \leftarrow r_{short} - i$;                                      dl_reduce.95
    fi                                                                             dl_reduce.100
od                                                                                 dl_reduce.105

$r_{short} \leftarrow \{[x \rightarrow \alpha \cdot] \mid x = x_0\}$;             dl_reduce.110
if $\exists\, r_j \in$ R s.t. $r_j = r_{short}$ then                              dl_reduce.115
    $k \leftarrow j$                                                               dl_reduce.120
else                                                                               dl_reduce.125
    $maxreduce \leftarrow maxreduce + 1$;                                          dl_reduce.130
    $r_{maxreduce} \leftarrow r_{short}$;                                          dl_reduce.135
    $R \leftarrow R + r_{short}$;                                                  dl_reduce.140
    $k \leftarrow maxreduce$;                                                      dl_reduce.145
fi                                                                                 dl_reduce.150
/* reduce via $r_k$ at readhead */                                                 dl_reduce.155
$default \leftarrow default + (r_k, readhead)$;                                    dl_reduce.160

$n \leftarrow |\alpha|$;                                                           dl_reduce.165
pop($n$ times);                                                                    dl_reduce.170
$q \leftarrow$ tos;                                                                dl_reduce.175
begin/* rewrite the input */                                                       dl_reduce.180
    $readhead \leftarrow readhead - n$;                                            dl_reduce.185
    $\rho_{readhead} \leftarrow x_0$;                                              dl_reduce.190
    $\rho_{readhead+1 \cdots readhead+n-1} \leftarrow \rho_{readhead+n \cdots readhead+n+n-1}$;   dl_reduce.195
end                                                                                dl_reduce.200
```

Figure 5.15: Default List Reduce Action and Deletion of Chain Rules

```
function sizeof (ρ) returns naturals ;                    sizeof.10
   declare ρ ∈ V;                                          sizeof.15
begin                                                      sizeof.20
   declare length ∈ naturals ;                             sizeof.25
   length ← 1;                                             sizeof.30
   while $\rho_{length} \neq \$$ do                        sizeof.35
      length ← length + 1;                                 sizeof.40
   od                                                      sizeof.45
   return (length − 1);                                    sizeof.50
end                                                        sizeof.55
```

Figure 5.16: Function sizeof

5.2. Analysis Implementation Tour

Analysis is implemented as a straight forward mapping into C of the definitions of sets and relations, and the algorithms presented in this report. Three global implementation strategies are followed. First, as much auxiliary information as possible is precomputed, usually in the most time efficient physical representation, although that may not be the most space efficient representation. This information assists the tests and construction that occur in the primary state iteration loop. Execution speed in this loop was deemed to be critical. Secondly, information describing states in the parser (the *next* and *action* matrices) generated by the state iterator is stored in a physical representation to minimize space, or is immediately written to the intermediate file. Thirdly, the algorithm performs blanket tests (§5.1.2). This tour first describes the static precomputed data, and then describes the dynamic structures, with code references and highlights interspersed.

5.2.1. Precomputed Data

Relations *leftfirst*, *rightfirst* and *follow* are used to detect nonuniformities, and construct the reduce itemsets. Logically, they are $|\mathbf{N}| \times |\mathbf{N}|$ boolean matrices. Physically, a relation is an array of pointers, each pointing to an array of words, containing enough bits for a relation row. All relation construction and relation algebra primitives are contained in file analrelat.c.

Items are constructed by first partitioning **P** into sets of syntactically identical productions. One such production is chosen as the *candidate* production, and the others are temporarily ignored. Items are represented as a dot position and a reference to the sourcing production. Item construction from rules, item manipulation and grammar rule manipulation are done in file analmakerel.c.

Itemsets are the principal data objects manipulated by the parser generator. Logically, they are an array of bits, one bit for each item. Physically, they have two representations. The first is speed efficient for all set operations, and space efficient for sets with many elements (high cardinality), and is the natural array of bits, implemented as an array of enough words. The second representation is a list of item indices with a terminator, and is space efficient for itemsets with few elements (low cardinality). Most kernel itemsets have only a few elements. The itemsets *firstset*, *partialset* and *instset* are items with initial, medial or terminal dots, respectively. The itemset *chaininstset* is the set of items with terminal dots sourced by chainrules.

An itemset vector is logically an array of itemsets, with one itemset for every symbol in **V**. Physically, an itemset vector is an array of pointers to the bitvector representation of itemsets. A specific itemsetvector is applied when that specific information about a given symbol is needed; itemsets described in the previous paragraph are applied when general information is needed for a blanket test. *Dotprecedes* is the set of items in which the dot precedes the given symbol, and is used in determining shifts in *shift.20*. *Non1stdotprecedes* is the same as *dotprecedes*, excluding initial items, and is used in *mindist.25*, *looprm.25* and *block.10*. The *addlist* for a given symbol is the set of items added by kernel closure for that symbol, and accelerates kernel closure in *close.10 – close.45*. For a given symbol, *redfollow* is a subset of all terminal items, in which that symbol can follow the left hand side grammar symbol of the rule sourcing an item in *redfollow*, and is used in *reduce.30* and *reduce.35*. *Lengthits* has as many member itemsets as the length of the longest right hand side of any production in **P** (and is not, by the above definition, an itemset vector). *Lengthits* is the set of all items with the sourcing rule having right hand side length less than or equal to a particular value, and is used in *expandR.185*.

Itemset primitives are contained in analitemset.c. These primitives include all set operations, itemset hashing functions, conversion routines between the two physical representations, itemset cardinality computation and itemset closure. Two things are special in analitemset.c.

First, the majority of tests in the algorithm test existence (or non existence) of a certain condition in the closed itemset associated with the state being considered. This test is made by seeing if a precomputed itemset that reflects all itemsets in which the condition holds (the blanket itemset) is disjoint from the tested itemset. Hence, the function *itsdisj* accounts for over 50% of the CPU time in the analysis phase; for the PDP11 example from Glanville's dissertation (Appendix A, pp 119), *itsdisj* is called 19247 times, and finds the argument sets disjoint 59% of the time.

Secondly, a numbering implementation trick is used to compute the kernel of a state shifted into (*shift.30*). If the item $[v \rightarrow \alpha \cdot x\ \beta]$ has index n in **I**, then the item $[v \rightarrow \alpha\ x \cdot \beta]$ has index $n + 1$. To compute this new itemset, all item indices must be incremented by one. In the bit vector representation, the increment corresponds to a bit vector left shift.

5.2.2. Dynamic Structures

The data structures describing states (**Q**) and the matrices *next* and *action* are dynamically extended each time a new state is generated. Further, each new reduce itemset dynamically extends the structure describing **R**.

5.2.2.1. Q

The state set **Q** is implemented as a list of state descriptors so states may be accessed in order. The unconsidered state iterator traverses this list. During the default list construction phase, **Q** is randomly accessed by an array of pointers to all state descriptors; this array is constructed after all states have been considered. **Q** must also be searched to find or insert a member itemset kernel, so the states are maintained on hash lists keyed by the itemset kernel. The algorithm in this report inserts new state descriptors at the end of the list implementing **Q**, and thus constructs and numbers the finite state machine (FSM) control graph of the parser by a breadth first search. In the early stages of writing **merge**, the author hypothesized that constructing the FSM control by a depth first search would number the states in such a way that would suggest a table compression heuristic. Constructing the FSM control depth first is done by inserting the new, unconsidered state descriptor in the state list immediately after the descriptor for the state being considered, so the new state is the next one considered. The parser constructed this way is isomorphic to the one constructed using breadth first search; the state numbering did not elicit any compression heuristics. In fact, the compressed tables for the two construction methods are identical. However, using the depth first construction method, the order in which states are considered is not the same as the order in which unconsidered states are created. This, together with the order in which attributes for considered states are written to the interphase temporary file, creates a referencing problem. The implementation assigns an ***internal state number*** to a state, which is the creation sequence number of an unconsidered state, and an ***external state number*** which is the state consideration sequence number. Internal state numbers are negative, and external state numbers are positive. For breadth first construction, external state numbers are the absolute value of internal state numbers; the first thing **merge** does is map internal state numbers to external state numbers, and the distinction is lost. The code that handles this search strategy is only a historical vagary, and should be exorcised.

5.2.2.2. *Action*

The *action* matrix tells the parser what to do when in state q on symbol v, and logically is dimensioned $|\mathbf{Q}| \times |\mathbf{V}|$. State indexing is provided from a pointer maintained in the state descriptor to a word vector; this vector is indexed by symbol. There are only a few unique rows in a typical action table (for the PDP 11 grammar from Glanville's dissertation, there are 198 states and 8 distinct action rows), so the table is stored as one physical row for each unique action row, with routines to lookup or insert a new action row. The entire action matrix is memory resident throughout the duration of this phase.

5.2.2.3. *Next*

The *next* matrix tells the parser which state to shift into, or which reduce set to apply, depending on the corresponding entry in the *action* matrix. *next* is the same size as *action*. For **shifts**, the entries in *next* are the internal state numbers of the appropriate itemsets in **Q**. For **reduce** actions, the *next* entry is the index of the reduce list associated with the reduce itemset, for shifts and reduce actions, respectively. There is no row duplicity in the *next* matrix (otherwise, two states would have the same itemset kernel), although many rows may differ only in a few symbols, a fact heavily exploited by **merge**. The row in *next* for a particular state is constructed when that state is considered, and is written to the interphase temporary file immediately after the state is fully considered. After all states have been considered, this file is reopened for random access reads, and in conjunction with a small cache of rows from *next*, is used in the default list construction phase which requires access to the entire *next* matrix. For the PDP11 grammar from Glanville's dissertation, the default list construction on 70 semantically restricted rules, accesses 154 rows in the *next* matrix (out of 198); with a cache size of 16, the cache hit ratio is 44%.

5.2.2.4. R

The reduce set **R** is implemented as a list of reduce descriptors. As with **Q**, the primary operation is to find or insert a member reduce itemset, so the descriptors of members of **R** are maintained on hash lists keyed by the itemset. Each reduce itemset is associated with a unique reduce list. This reduce list is constructed by considering the longest rule(s) sourced by an item in the reduce itemset. The rules from P syntactically identical to the candidate rules used to make the itemsets are then sorted by decreasing semantic restriction, and formed into the reduce list. This construction is carried out when a reduce itemset is inserted into **R**. Possibly, a unique default list is associated with the reduce list when all members of the reduce list are semantically restricted. The reduce list is a negative terminated list of grammar rule indices; if the negative terminator on a reduce list is less than −1, it is the negative index of the default list associated with the reduce list. (See the domain hack in *expandR.445*.) Both reduce and default lists are allocated dynamically, are memory resident throughout the entire analysis phase, and are concatenated together by the output routines into their final form as arrays.

5.3. Analysis Implementation History

William Joy wrote the parser generator found in **analysis**, as well as the library routines manipulating itemsets and relations. The author added the code to construct reduce lists, to remove loops and to construct default lists, as well as all I/O interfaces to **tmdl** and **merge**. This was the last piece of code that William Joy wrote in the CGGWS; the author wrote the remaining phases **merge**, **ccode**, and **codegen**.

6. Merge

Merge is one of two physical phases applied after **analysis** and before **codegen**. **Merge** modifies the physical representation of the parse tables, but not their virtual structure. **Merge** performs the housekeeping functions of merging into a single file the three output files created by **analysis** and the single machine description file created by **tmdl**. **Merge** applies an **analysis** supplied mapping function between internal state numbers and external state numbers. **Merge** performs consistency checks, and formats the parser's tables for printing. Most importantly, **merge** substantially compresses the parse action and next tables; for the glan11 example, the tables are compressed to 18% of their original size.

Merge does a physical compression that trades code generator space for code generator time. The code generator only spends a small fraction of its time doing table lookup, so this trade-off costs only a little. The tables are represented by lists, requiring list traversal instead of direct indexing to access an element, but lists share common suffixes and exploit regularities. Figure 6.1 shows some of the compression statistics; the meaning of some rows in the figure will become apparent later.

The table compressor in **merge** knows that the matrices being compressed are parser tables, but does not exploit any properties of the machine description grammar that **analysis** didn't already use to build the tables. Table compression heuristics predominately view the parser tables as tables of elements that are either equal or unequal. Since the tables are regular, these heuristics do very well. The uncompressed tables let the parser immediately detect syntax errors in the IR, but the compressed tables slightly delay error detection, since minor assumptions are made about rows containing only **reduce** or **error** entries. Since the IR is assumed error free, and the parser has no syntactic error recovery anyway, it is no great loss to delay error detection. Since **merge** is run relatively infrequently, its inefficient paging behavior is tolerated.

In the following description, we assume that the logical action and logical next tables have been overlayed into one table. (This is indeed the case.) Recall that there is a one to one correspondence between rows of the table and states in the parser (both terms will be used interchangeably, depending on the context). There is also a one to one correspondence between columns in the table and symbols in **V**. λ (alternatively, \$ or $\bullet$) is a root symbol, as are all $v \in \mathbf{V}$ s.t. λ FIRST v.

6.1. Row Analysis of the Parser Tables

Disregarding error entries, the rows in the table can be grouped into five categories:

6.1.1. Row 0

This row is special, since it is the only row with an **accept** action, and the only row in which root symbols **shift**.

6.1.2. Unreachable Rows

The loop removal algorithm in **analysis** may have split a state and created a new state that is **shift**ed into instead of the original. The original state may then be

Compression Statistics for Ten Machine Descriptions	glan11	11	full11	nc11	glan360	Z8000	8086	3000	VAX	NAND
Initial										
# rows	197	782	1346	1406	193	316	936	672	1808	65
# unreach	1	1	1	1	2	1	1	0	0	0
# discard	53	119	211	155	50	109	116	102	43	39
lg init	14K	80K	211K	144K	13K	23K	69K	54K	134K	2.4K
Compress										
# slices	222	470	1793	1996	236	471	1255	897	2547	41
# overlayed	381	1805	5034	3326	379	692	3475	1704	6106	142
lg compr	2.5K	12K	24K	23K	2.6K	4K	17K	11K	33K	.6K
Factor (%)	18	15	12	16	20	17	24	20	24	26
Symbols										
# syms	27	43	70	43	26	29	29	32	29	10
lg sym	432	688	1120	688	416	464	464	512	464	160
Dlists										
lg dlist	248	1580	2372	3196	8	464	1860	175	5148	200
Rlists										
lg rlist	1186	3742	6256	6278	744	1894	4548	2430	5664	560
Rules										
# rules	101	319	516	525	63	161	342	182	463	44
# bundles	432	1.9K	3.2K	4.1K	253	647	2.5K	1.2K	5.9K	122
lg rules	8.4K	34K	54K	66K	5K	13K	42K	21K	85K	2.8K
Assembly										
lg assy	1.5K	5.7K	9.3K	10K	1.0K	2.0K	5.9K	2.7K	14K	191
Total										
lg total	14K	57K	98K	109K	9.8K	22K	71K	38K	143K	45K

Key:

K	1000
lg	size in bytes
#	number
%	percent
compr	compressed
sym	symbol
rlist	reduce list
dlist	default list
assy	assembly list

Figure 6.1: Compression Statistics for Ten Machine Descriptions

unreachable, and should be discarded.

6.1.3. Reduce–Only Rows

These rows **reduce** on all symbols. Some of these rows **reduce** using the same $r \in \mathbf{R}$ for all symbols, while others **reduce** by $r \in \mathbf{R}$ for non root symbols and by $r' \in \mathbf{R}$ for root symbols. Rows of the first kind can be made unreachable by turning all **shift** entries into this row into a new action, **shift_reduce** by $r \in \mathbf{R}$. This new action forces the parser to **shift** one symbol and then **reduce** by r, possibly delaying error detection.

6.1.4. Shift and Reduce Rows

Typically, the **reduce** entries are in root columns, although not always. For example, in the glan11 example, the production

$\lambda \rightarrow$ ^ + k.1 r.1 k.2 "mov k.2, *k.1(r.1)"

was purposely introduced as the only instruction with indexed deferred addressing, as discussed in Glanville's dissertation on page 90. Hence state 116 in Figure 6.3, corresponding to state 166 in Glanville's dissertation, **shifts** only on k, and **reduces** on all other symbols.

6.1.5. Shift-Only Rows

All entries in this row **shift**.

6.2. Column Analysis of the Parser Tables

Ignoring **error** entries, and row 0, column vectors in the table either **shift** only, or **shift** and **reduce**. **Error** only columns are possible if a symbol was defined but never used in the grammar. **Reduce** only columns are impossible. In order for a symbol to cause a reduction, it must have been included in the right hand side of at least one production. Consequently, at least one state will **shift** that symbol. A key observation about most columns, exploited by the compressor, is that most **shift** entries **shift** into the same state. Here is why. Consider all states $q \in \mathbf{Q}$, containing in their core the item $[u \rightarrow \alpha \cdot x\ \beta]$ where $x \in \mathbf{N}$. (Since x is a non terminal, it is a register.) Then, if x FIRST v, and the item $[w \rightarrow \cdot v\ \gamma]$ is also in q, ACTION$[q, v]$ = **shift**, NEXT$[q, v] = q'$. If the symbol v is only used in the grammar in a production of the form $x \rightarrow v\ \delta$, then all of the shift entries in column v will be the same. The greater the number of grammatical contexts for v, the more times the column entry for v differs from **shift** q'.

6.3. Column and Row Definitions

A *slice* of a row $r_1 \cdots r_n$ is a sub row of contiguous elements, $r_i..r_j$, $1 \le i \le j \le n$. Slices always refer to rows of the action/next table. (The term slice comes from the ALGOL68 lexicon.) The *per column modal shift entry* is the shift entry that occurs most frequently in that column. The *column activity* is the number of shift entries in that column different from the per column modal shift entry; the *row activity* is the number of shift entries in that row differing from their respective column modal shift value. The *canonical shift slice* is a full length row slice constructed from the modal shift entry for each column.

There is an intuitive correlation between activity and instruction semantics. Symbols with high activity are those occurring frequently in the instruction grammar and express primitives common to computers; on the PDP 11 the most active symbols are: "r" (general integer register), "k" (integer constant), "~" (indirection) and "+" (integer addition). Symbols with low activity occur only in certain productions: on the PDP 11 they are "*" (multiply), "/" (divide) and "%" (modulo), as expected.

6.4. Row Reordering

The initial row enumeration order is determined by the way **analysis** constructed and considered states, while the initial column ordering is the same as the relatively arbitrary symbol declaration order in the TMDL for the instruction grammar. Hence, the rows and columns of the table can be reordered, provided all extant implicit references to rows and columns are changed. The table itself (stored as a row major matrix) need not be shuffled if the row and column mapping functions are applied at each access, but it is convenient to do so, both for later efficiency gains, and for logical clarity in **merge**'s sub phases. Reordering rows is fairly painless, but reordering columns on a paged virtual memory machine is expensive. However, global column reordering is necessary to slice rows efficiently for overlaying.

Row reordering must discard the unreachable states, and preserve row 0 as row 0, but since the compression is done on a per row basis, other row reordering is done only to improve the visual aesthetics of the printed, reordered table. Rows are separated into **shift** only, **shift_reduce**, and **reduce** only partitions. Each partition is further divided depending on the number of columns **shift**ing or **reduce**ing, and further by the row activity.

6.5. Column Reordering

Column reordering first tries to maximize, over all rows, the length of **error** slices, or **reduce** only slices. This immediately partitions the symbols into three groups: root symbols, *primary symbols* and *special symbols*. Special symbols are known to occur only in certain contexts; on the PDP 11, these include label references, label definitions, condition code references and the compare operator. Root symbols are V − { primary symbols } − { special symbols }.

Within the partition of primary symbols, columns are reordered by decreasing activity. For each row, this tends to create long slices of shift entries differing slightly from the canonical shift slice.

6.6. Graphically Interpreting Reordered Tables

Viewing the table as a whole often helps intuitively measure the uniformity of the instruction grammar. Large rectangular blocks completely filled in with either **error**, **shift** or **shift_reduce** entries, or with **error** or **reduce** entries indicate a uniform grammar. Small, patchy blocks isolated together, or stand alone columns, should be viewed with suspicion, unless they are in (row, column) areas of the table corresponding to the parser's context when analyzing special, idiomatic constructs guaranteed to be the same in the IR. Indeed, when a partial grammar for the INTEL 8086 was developed, the blocking report generated by **analysis** was turned off (as is normally the case), but grammar irregularities associated with the index register model were discovered by table inspection. To show the similarities between rows, the first few rows for the action and next table for the *glan11* grammar appear literally in Figure

6.2. To show the effect of column reordering, the entire *glan11* action table

```
          k         r         ~         +         o         e         -         d         *
    -------------------------------------------------------------------------------------------
  0|                                                                                          |
  1|                                                                                          |
  3|                                                                                          |
  9|                                                                                          |
 15|s:  83    s:  85    s:  38    s:  57    s:  79    s:  81    s:  67    S:* 1     s:  15    |
 16|S:   1    S:   6    s:  34    s:  61    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
 17|s: 122    S:   6    s:  34    s:  53    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
 18|S:   1    S:   6    s:  34    s:  56    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
 20|s:  72    s:  74    s:  18    s:  47    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
 21|S:  22    S:   6    s:  34    s:  55    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
 22|s: 100    S:   6    s:  34    s:  45    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
 23|S:   1    S:   6    s:  34    s:  58    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
 24|s: 100    S:   6    s:  34    s:  59    S:* 2     S:* 3     s:  67    S:* 1     s:  15    |
    -------------------------------------------------------------------------------------------

          /         &         |         !         m         .         =         j         :
    -------------------------------------------------------------------------------------------
  0|                                                        ACCPT s:  20    s:   1    s:   2  |
  1|                                                                                          |
  3|                                                                                          |
  9|                                                                                          |
 15|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 16|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 17|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 18|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 20|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 21|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 22|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 23|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
 24|s:  46    s:  69    s:  66    s:  70    s:  65                                            |
    -------------------------------------------------------------------------------------------

          <         >         L         G         E         N         l         c         ?
    -------------------------------------------------------------------------------------------
  0|s:   4    s:   6    s:   7    s:   5    s:   8    s:   3                                  |
  1|                                                            S:  68                        |
  3|                                                            s:  13                        |
  9|                                                                      S:  69    s:  68    |
 15|                                                                                          |
 16|                                                                                          |
 17|                                                                                          |
 18|                                                                                          |
 20|                                                                                          |
 21|                                                                                          |
 22|                                                                                          |
 23|                                                                                          |
 24|                                                                                          |
    -------------------------------------------------------------------------------------------
```

Key

A	accept
blank	error
s	shift
S	shift_reduce
R	reduce

Figure 6.2: Table Entries for First Few Rows for *glan11* Grammar

schematically appears in Figure 6.3, with adjacent identical rows compressed together to save space on the page.

6.7. Row Decomposition

Once the table has been reordered, the rows can be easily decomposed into slices on a row by row basis. Row decomposition and compression immediately follows. All decomposition is done within one row. A row is decomposed into slices, isolating adjacent all-same row entries into a slice, and breaking adjacent **shift** entries into 2 slices, one of which can overlay the canonical **shift** slice, and the other which can not. Since the global column reordering by activity empirically works well on each row, the first **shift** only slice (that is overlayed with the canonical row) tends to be long. Slicing does not become so fine that the slice descriptor is bigger than the slice.

```
      kr^+oe-d*/&|!m.=j:<>LGENlc?              kr^+oe-d*/&|!m.=j:<>LGENlc?
     ----------------------------             ----------------------------
    0|              Asssssssss   |       47-49|ssssSSsSssssss             |
  1-2|                        S  |          50|SsssSSsSssssss             |
  3-8|                        s  |       51-64|ssssSSsSssssss             |
 9-14|                         Ss|       65-69|SSssSSsSssssss             |
   15|sssssssSssssss             |          70|SSssSSsSssssss             |
   16|SSssSSsSssssss             |       71-72|ssssSSsSssssssRRRRRRRRRR   |
   17|sSssSSsSssssss             |          73|SSssSSsSssssssRRRRRRRRRR   |
18-19|SSssSSsSssssss             |          74|ssssSSsSssssssRRRRRRRRRR   |
   20|ssssSSsSssssss             |       75-79|SSssSSsSssssssRRRRRRRRRR   |
   21|SSssSSsSssssss             |          80|ssssSSsSssssssRRRRRRRRRR   |
   22|sSssSSsSssssss             |       81-83|SSssSSsSssssssRRRRRRRRRR   |
   23|SSssSSsSssssss             |          84|ssssSSsSssssssRRRRRRRRRR   |
   24|sSssSSsSssssss             |       85-87|SSssSSsSssssssRRRRRRRRRR   |
   25|SSssSSsSssssss             |          88|ssssSSsSssssssRRRRRRRRRR   |
   26|sSssSSsSssssss             |          89|SSssSSsSssssssRRRRRRRRRR   |
27-31|SSssSSsSssssss             |          90|ssssSSsSssssssRRRRRRRRRR   |
32-33|sSssSSsSssssss             |          91|SsssSSsSssssssRRRRRRRRRR   |
   34|SSssSSsSssssss             |       92-94|SSssSSsSssssssRRRRRRRRRR   |
   35|sSssSSsSssssss             |       95-98|SsssSSsSssssssRRRRRRRRRR   |
   36|ssssSSsSssssss             |      99-102|SSssSSsSssssssRRRRRRRRRR   |
   37|SSssSSsSssssss             |         103|SsssSSsSssssssRRRRRRRRRR   |
   38|SsssSSsSssssss             |         104|SSssSSsSssssssRRRRRRRRRR   |
   39|SSssSSsSssssss             |     105-108|SsssSSsSssssssRRRRRRRRRR   |
40-43|ssssSSsSssssss             |     109-110|SsssSSsSssssssRRRRRRRRRR   |
   44|SsssSSsSssssss             |     111-114|SSssSSsSssssssRRRRRRRRRR   |
   45|ssssSSsSssssss             |         115|SsssSSsSssssssRRRRRRRRRR   |
   46|SSssSSssssssss             |         116|SRRRRRRRRRRRRRRRRRRRRRRR   |
                                       117-142|RRRRRRRRRRRRRRRRRRRRRRRR   |
```

Key	
A	**accept**
blank	**error**
s	**shift**
S	**shift_reduce**
R	**reduce**

Figure 6.3: Table Schema for *glan11* Grammar

The sliced row is then represented as a list of slice descriptors, where each slice descriptor contains a lower and upper symbol bound, and an index into the area where slices are concatenated together. Slices with length one, or with entries doing the same thing, are trivial slices represented fully in the slice descriptor. Isolated non trivial slices are either overlayed onto the canonical shift slice, or appended to it. Row slice descriptor lists are then internally ordered to share tails, and are eventually represented as indexed elements in a slice descriptor vector, with table entry for row i provided through the ith element in the vector.

6.8. Other Table Compressions

Merge makes no attempt to compress the action and next entries into their smallest possible bit fields, nor does it attempt to improve the representations for default lists, reduce lists or assembly strings. These compressions were deemed to be outside of the scope of the project. For the glan11 example, these lists and strings occupy 95% of the total machine description storage space! However, fields in the various descriptors have been laid out with care so that both packing and access using byte fetches is efficient. **Merge** does pack ACTION into 3 bits of a 16 bit word, with the NEXT entry (always an index) stuffed into the remaining 13 bits. Hence, figures given in Figure 6.1 are larger than Glanville gives, as he assumed the smallest possible bit field for all values. More complicated sequential coding schemes are not suitable because the tables are randomly accessed.

7. Ccode

Codegen was initially written to accept parsing, symbol and rule semantic tables directly from **merge**. Codegen initialized its tables in a slow and complicated manner, reflecting both the necessary consistency checks, and the complexity of the intermediate form and **codegen** data structures. For the PDP11 grammar, **codegen** would take 3 seconds of VAX CPU time initializing its data space before any IR was read. While this initialization was excessive for production runs of **codegen**, **codegen** could be flexibly developed together with the previous phases without worrying about another table filter.

When **codegen** became relatively stable and began to be exercised with many test programs for many machines, **ccode** was written from the **codegen** initialization routines. **Ccode** is a filter casting the tables into preinitialized "C" data structures; these are then compiled into tables that **codegen** can use without suspicion and loading overhead. In addition, the majority of tables are loaded into read only (text) space.

Ccode is heavily dependent on the layout and naming conventions used throughout CGGWS. Hence, modifications to **codegen** and phases preceding **ccode** may have to be reflected in **ccode**'s output routines. In addition, the parse tables are arrays of unions, each union defined from structures in turn defined with bit fields. Each structure has an aggregate size after packing of exactly two bytes, which is the union size. Since "C" does not allow union initialization, a size compatible structure is defined, and an array of such structures initialized, and then aliased through an assembly language loophole to the desired array of unions. In addition, **ccode** and the following compilation are very slow (typically 61% of the code generation construction time) because the C compiler is slow with initialized data, and there is a tremendous quantity of it. This increases the development iteration time for a new machine description/code generator, since **ccode** can not (now) be dropped from the iteration loop. However, during the last months developing CGGWS, **ccode** and the table structure it implements were stable. **Ccode** was abstracted into a black box, even though some changes, fortunately not affecting data structures, had to be made in both **analysis** and **ccode**.

8. Codegen

Codegen is run once for every program compiled into code for the target machine. **Codegen** consists of a set of standard modules interpreting the tables constructed by **analysis**, **merge** and **ccode**: these tables are loaded into read only storage to reduce **codegen** invocation overhead. **Codegen** performs seven major tasks: IR scanning and tokenizing, IR semantic checking, IR parsing and target machine instruction pattern recognition, production semantic matching, default list application, and register allocation. Because **codegen** performs only one pass over the IR, not only is the register allocator overly complicated, but the resulting code can be time and space inefficient for machines with complicated, non uniform register architectures or for complicated IR which has not had prior optimization or analysis.

The tour through **codegen** will discuss the six major tasks it must perform. Included in this discussion will be examples taken from real machines. These examples highlight the difficulties **codegen** might have with a particular target machine architecture, or with a particular IR example; in the whole, these examples are an incomplete cookbook for writing machine descriptions, and suggest what **codegen**, or an improved successor, should do differently.

8.1. IR Scanning and Tokenizing

Codegen recognizes essentially the same internal representation used in Glanville's dissertation. Since the IR is a sequence of characters, the IR scanner is relatively inefficient (typically, 9% of the execution time) but the IR can be easily generated by hand as an aid to debugging. The scanner in **codegen** shares **tmdl**'s scanner, ensuring identical lexical conventions.

8.1.1. IR Comments

The scanner recognizes two forms of comments. The first form is delimited as in "C" by "/*" and "*/". The contents of these comments may extend across lines, can contain a "$" or "$no" prefixed **codegen** option to toggle on or off, much as the PASCAL 6000 compiler directive comments are enabled. The remainder of these comments are ignored. The second form is delimited by double quotes, may not extend across a line, and is stripped of delimiters and copied directly to the assembly language output file when the comment is encountered. These *code comments* are intended to allow the IR generator to place comments in the assembly code, and to access assembler primitives, though such access makes the IR generator non portable. In the assembly language file, the position of the code from code comments in relation to the code emitted for IR is only guaranteed if the code comments appear between the trees in the IR.

8.1.2. IR Tree Recognition

The IR scanner never reconstructs an explicitly linked forest of binary trees that is implicit in the prefix flattened IR; **codegen** does not do any tree manipulations, so only needs the flattened form. However, the scanner does know where individual trees start in the IR stream. This information is used by the symbol table, the synchronous reader and the register spill algorithm. Let $E = e_1 \cdots e_n$ be a sequence of symbols in **V**, e_1 a root level operator. Let $W(e)$ be 1 if e is an operand, otherwise let $W(e)$ be 1 − the number of operands for e. Then, the ***Polish Prefix level*** (or simply ***level***) for E is $\sum_{1}^{n} W(e_i)$. By the definition in Glanville's dissertation, page 34, the level of a complete Polish prefix expression is 1, and for prefixes of prefix expressions the level is always less than one.

8.1.3. Synchronous Reading

The scanner implements synchronous reading to ensure that all code for an expression tree is emitted before the code comment following that expression tree is copied out. Without synchronous reading, when the parser required one token of lookahead after having shifted the last symbol in a tree onto the parsing stack, it would request the scanner to read the root symbol of the next tree. The scanner would skip across and copy out a potentially unbounded number of code comments before it found the root symbol, at which point the remainder of the expression code would be emitted. This mis-synchronization is undesirable for the IR generator. Instead, when an entire tree has been read and another token requested by the parser, the scanner returns a preliminary end of file (in alternative notations, "$", "." or "λ"). The parser is then restarted from state zero if there are more trees in the IR forest.

8.1.4. Symbol Table

All symbols and their binding values encountered in the IR are looked up or entered into the symbol table. **Tmdl** has initialized and locked into the table the IR symbol and bundle qualifiers. The table is purged of non-locked symbols after each complete IR tree has been parsed, since the grammar, organized around root symbols, ensures that there are no inter tree symbol comparisons. (Descriptions for common sub expressions are not maintained in the symbol table, as they can have lifetimes exceeding that of a single tree.) In **codegen**, most semantic attributes are compared by comparing symbol table pointers for equality. The values in the table are not compared, akin to the difference between LISP *eq* and *eql*. This is done in the interest of speed, but means that "00" and "0" are not the same constants, although they both have the same value (zero). If the IR generator ensures that only one external representation of a value is allowed (perhaps by always using the same output formatting routine), and the TMDL author uses the same external representation in the TMDL, then this **codegen** time saver will be no problem.

8.2. IR Semantic Checking

The scanner checks if the type and value semantics for attributes bound to symbols in the IR are correct. In this implementation, only IR operands (the leaves of the IR tree) may have a binding. Future implementations may allow operator bindings to implement rudimentary operator grouping. Leaves in the IR tree may only be ranged constants or registers, and must have a "." qualified binding to a particular value. For constants, the value of the binding may be either a symbolic name, in which case no range or type checking is done, or a character, integer or floating constant with "C" syntax and semantics. In the latter case, the type of the symbol is inferred from the constants used in the TMDL definition of that symbol, and that type is compared with the type of the constant bound to it in the IR. The value is also compared to see if it is in the proper range. For registers, the value of the binding must be the symbolic name of a dedicated register. Any semantic violations, including IR syntactic ill-formedness as determined by a prefix level > 1, are non recoverable errors, indicating that the IR generator is at fault.

8.3. IR Syntactic Recognition

A table driven shift-reduce parser, driven from the precomputed Action and Next matrices, syntactically recognizes instruction patterns in the IR token stream. This parser is essentially the same as that from Glanville's dissertation (Algorithm 4.1, page 50), augmented with a **shift_reduce** action. The parse stack contains the state record needed by the parse algorithm, the vocabulary symbols shifted onto the stack, and their associated semantics from the IR or the register allocator. An IR token stack (not to be confused with the parse stack) isolates the IR scanner from the parser, and provides a repository for the left side of a reduction and for tokens removed from the parse stack before a reparse. (see §8.10). The parser drives the majority of **codegen**. All register allocation, semantic checking and code emission is done after the parser has found a *handle* residing on the top of the stack. Because the handle is used in the context of code generation, and since the top elements of the stack are manipulated by the default list application code, the handle is called the *matched instruction pattern*, or simply the *pattern*. The pattern is indexed from 1 to n, and $pattern_i \cdots pattern_j$, $1 \le i \le j \le n$ will be called a *sub–pattern*. Sub–patterns are referenced in default lists, where they are recursively treated as patterns.

8.4. Production Semantic Matching

Semantic matching must simultaneously find the most semantically restricted production from the candidates on the reduce list, and try to satisfy register allocation constraints. The semantic matcher must choose productions carefully, as once the choice is made, the register allocator considers only that production, and may generate terrible code. For complicated machine descriptions with many idioms, semantic matching can be a time consuming critical inner loop because the reduce list presorted by semantic restriction is linearly searched. For each production, a number of semantic attributes must be checked. (For the PDP 11 grammar there are 354 productions, having 388 total restrictions, or 1.09 restrictions per production. There are 294 reduce lists, with an average length of 2.36.) The more idioms in the grammar, the longer the reduce lists tend to be; differences between candidates are not isolated from similarities, so many comparisons are repeated unnecessarily. **Tmdl** and **analysis** have preconstructed the reduce list and the rule descriptors with semantic matching in mind, but such aids increase the space required for rule descriptors so they take up roughly 60% of the total table size. However, experiments with machine descriptions having only a few idioms included in the grammar show that the semantic matcher accounts for less than 1% of **codegen**'s execution time. Consequently, a more complicated encoding scheme could be used. Space could be saved if rule descriptors were factored and shared, much as **merge** did for the Action and Next matrices.

The semantic attributes associated with symbols in the pattern are pointers understood by the register model, or pointers understood by the symbol dictionary; in either case only one efficient pointer comparison for equality needs to be made.

Even if the semantics required by a a restricted production match those in the pattern, a less restricted production may emit better code in the long run, but only because the semantics are incomplete in this implementation. Consider again the VAX's integer register to register add instruction, first discussed in §3.1.4.1. Assume the production

$$r.1 \rightarrow + r.1\ r.2$$

was chosen first (it has the highest semantic restriction value); and the registers bound to both r.1 and r.2 had future uses. The register allocator, stuck with this overly restricted production, would first copy the contents of the register currently bound to r.1 into another register, and then use the new register in this production. If instead the production

$$r.1 \rightarrow + r.2\ r.3$$

were chosen, knowing the current and expected register allocation, this extra move would be avoided. Consequently, if the semantic matcher finds a production whose use would entail a move to protect a register, it will consider other less restricted productions on the reduce list. This example suggests that instead of a static ordering, there should be a way to weight the selection based on register use counts[1].

6.5. Default List Application

If there is no production semantically matching the pattern, **analysis** will have constructed a default list. The pattern is isolated into the sub–pattern specified by the base and length information from the default list. Semantic matching, register allocation and code emission are performed recursively for the sub–pattern. When code for the sub–pattern is emitted, the pattern is collapsed together (as the sub–pattern may have been internal to the matched pattern), and the next sub–pattern on the default list isolated. This is exactly the same rewrite performed in the default list construction algorithm, §5.1.14. For an example taken from the PDP 11, if the IR input to the code generator were:

$$= + k.2\ r.6 + \uparrow + k.4\ r.6\ k.1$$

then it would syntactically match the production:

$$\cdot \rightarrow\ = + k.1\ r.2 + \uparrow + k.1\ r.2\ k{=}1 \quad \text{"}\mathbf{inc}\ \text{k.1(r.2)"}$$

but would semantically mismatch because the semantic values on the constants are not the same. The default list for this pattern consists of:

[1]Alternatively, TMDL could be extended to include more comprehensive semantic attributes or predicates to be met before a rule could be applied.

seq	place	Reduce List/[Default List]	assembly string
1	6	"r.1 → ↑ + k.1 r.2"	"**mov** k.1(r.2),r.1"
2	5	"r.1 → ↑ + r.1 k=1"	"**inc** r.1"
		"r.1 → ↑ + r.1 k.1"	"**add** $k.1,r.1"
		[no default list]	
3	1	"dot → = + k.1 r.2 r.1"	"**mov** r.1,k.1(r.2)"
		[no default list]	

The default list is applied in sequence:

step	*stack*										*sub pattern*	*collapse*
	1	2	3	4	5	6	7	8	9	10		
1	=	+	k.2	r.6	+	↑	+	k.4	r.6	k.1	6..9	10 to 7
2	=	+	k.2	r.6	+	r.0	k.1	s	s	s	5..7	none
3	=	+	k.2	r.6	r.0	s	s	s	s	s	1..5	none
4	λ											

producing the code:

```
mov   4(sp), r0
inc   r0
mov   r0, 2(sp)
```

Note that there were no recursive applications of default lists, as all reduce lists on the default list for the memory increment instruction were without default lists. (Recursive application of default lists never happens on the PDP 11.) In step 1, there was a sub pattern internal to the longer pattern, necessitating a collapse of the stack. In step 2, if the constant were not 1, then we would have generated the more general, non idiomatic add instruction.

8.6. Assembly Code Emission

Tmdl produced a massaged assembly code prototype for each production. The assembly string is scanned to isolate the embedded bundle links. These bundle links are extracted and replaced by the external print representation of the corresponding semantic attribute found in the pattern. The largest fraction of **codegen**'s time is spent constructing assembly strings.

8.7. Register Allocation

The register allocator performs all of the tasks outlined in Glanville's dissertation §4.3, working from a register model simulating the contents of the registers during execution of the code emitted as a side effect to each reduction. In the following discussion, the phrase "during execution of the code emitted as a side effect to" is dropped from most contexts where it should appear. Further, the term "register" and the value that register contains at execution time are used interchangeably.

8.7.1. Register Allocation Assists by the TMDL Grammar

Each reduction to a non-λ non terminal requires one and only one destination. The destination may be a register or a register pair. The semantics of TMDL and fundamental restrictions in the parser do not allow two (or more) destinations to be specified. For example, one can not "track" side effects on the condition code or the effect of a register auto increment in parallel with the primary result of the instruction. Hence, the PDP11 divide instruction, which produces both the quotient and the remainder, can not be completely modeled in TMDL (see §3.2). To simulate two simultaneous destinations, one can use grammatical tricks such as defining new non terminals representing the power set of all register classes, together with an augmented grammar. In the worst case, these tricks can exponentially increase the table sizes and grammar complexity, although in practice the grammar could be augmented with the few that occur, together with a selection pseudo production. The grammar can be modestly augmented to save register allocation information in the parser's state. For example, the IR assignment operator, "=" is only used in the Glanville's dissertation as a root operator. On the PDP11, any instruction implementing the "=" operator sets the condition codes reflecting the new value of the destination. If the IR were extended to allow "=" as a non root symbol, then the PDP11 grammar could be augmented with additional productions of the form:

c.1 → ? *assignment tree* k=0 "*code for assignment tree*";

where *assignment tree* is a tree rooted with "=" implemented by one instruction. For the glan11 grammar, there are 33/101 assignment tree productions, so that grammar would be augmented by 33 new productions.

8.7.2. Allocating a Destination

The single destination for a production is found before the right hand side of the production is popped from the parsing stack. The production isolated and waiting for a destination is said to be *pending*. This destination can come from one of three places. The destination may be semantically bound to a non terminal on the right hand side, implying reuse. Or, the destination may be captured from other soon to be freed registers on the right hand side, the register may come from the free register pool, or the free pool must be extended and then used. After the instruction is emitted, registers from the right hand side with no future uses are reclaimed. If a selected destination has future uses and can not be overwritten, an intra–register move must be done to protect the dedicated register.

8.7.3. Register Trauma

The register allocator is prepared to handle the contingency when more registers are needed than are available. When this happens, registers must be *spilled* and *unspilled* into temporary storage. **Codegen** does not assume a register preestimation pass over the IR tree (for example, Sethi–Ullman numbering), nor does it assume that the IR tree has been rewritten and decomposed in a machine dependent or independent way to avoid unexpected spills. The code generator does only one pass, left to right, over the flattened IR, and has only a feeble oracle to consult regarding future IR complexity that might bias the register allocation performed in the present. S.C. Johnson's affirmation[2] about the Portable C Compiler, which *does* do rewrites to avoid unexpected spills is, in retrospect, well taken: "[This compiler feature] is one of the more significant, successful departures from most other

[2] *A Tour through the Portable C Compiler*, S.C. Johnson, Bell Telephone Laboratories, page 15. (no date).

compilers. It means the code generator can operate under the assumption that there are enough registers to do the job, without worrying about temporary storage". The complex, non modular and buggy parts of **codegen** are handlers and contingency planners for the traumatic register spill and unspill. **Codegen** could be cleaned up both by totally rewriting it (having learned what works and what does not work), or, as Johnson does, by writing a phase that rewrites the input trees and thus avoids register spills.

To facilitate spilling, the register model is broken into parts handling physical and virtual registers. We now discuss these two models and their interaction.

8.7.4. Physical Registers and Register Classes

The TMDL **$register** section defines which registers are available on the target machine, and partitions the registers into allocatable and dedicated sets. The non terminal declarations in TMDL establish other register classes, corresponding one to one with non terminals, and assign registers singly or in pairs to these classes. The implementation of the register model places a register descriptor on a list for each class to which it belongs, with separate lists for allocatable and dedicated registers. This list can be searched quickly to find a free register in a particular class. The lists traversed are typically short because only a few registers must be examined before a free register is found. The lists are not dynamically reorganized to improve search time.

Physical registers may be:

free	A free register may be allocated to any register class it belongs.
temp	A temporary register contains an intermediate result computed by code emitted when the grammar reduced to a non terminal.
cse	The value in a cse register is a common sub expression, §8.8. Its properties are intermediate between temp and dedicated registers.
dedicated	The value in a dedicated register may be read but not written.

Consistent with TMDL declarations, double registers have their own register class(es) in which they are indecomposable, so they may be allocated, spilled and used as common sub expressions just like single registers. Because **codegen** can spill single or double registers, and since it has no lookahead, traditional packing problems are non existent. **Codegen** has a special case for double registers in both the register allocator and the register spill routine. The allocator examines successively more spill-expensive choices, moving from (0 used, 0 free) to (1 used, 1 free) and finally to (2 used, 0 free). The latter case requires a double register spill. A double register decomposed in the destination of a pending reduction is semantically indecomposable for all register allocation purposes until all assembly code is produced. Then, **codegen** decomposes the register into its constituent saved and discarded single registers, depending on the reduction semantics. (See §3.2.)

8.7.5. Virtual Registers

Virtual registers represent values in a particular register class that have been computed, reside someplace, and will be used sometime in the future. For every distinct value in existence, there is a distinct virtual register. All references to registers, especially semantic attributes of symbols on the parsing stack, are through virtual registers. It is the function of the virtual registers and the spill/unspill algorithm to simulate an unlimited supply of physical registers. This is analogous to virtual memory systems:

register allocation	virtual memory
physical register	page of primary memory
virtual register	page descriptor
spilled register	page on secondary memory
spilling	page out
register use	page hit
unspilling	page in
unspill algorithm	page replacement algorithm
dedicated physical register	unpageable system area
register with future uses	read only page
terrible register use	thrashing

The analogy is not complete, however. With register allocation, one knows which registers have how many future uses, although one does not know when they will be used. Registers must be allocated within classes; most VM systems do not distinguish pages based on contents (other than a read only distinction). Finally, the structure of the IR, and the way registers are referenced, gives different kinds of information to the unspill algorithm than is available to an LRU page replacement algorithm.

A new virtual register is created when a value is computed by a reduction, although a virtual register may be reincarnated if a register on the right hand side is used as the destination. Immediately after the reduction, the virtual register is bound to the physical register actually holding the value. Virtual registers and their values (and associated storage) are alive until the virtual register's reference count falls to zero. Dedicated physical registers are permanently bound to immortal virtual registers; at creation physical registers holding common sub expressions have use counts which are the number of uses of the common sub expression, and expression temporaries have a use count of one. Free physical registers are not bound to any virtual register.

Without spills, values will stay in the physical register until the virtual register is last used. If the physical register is spilled, the virtual register is bound to a spill descriptor containing spill information. Whenever a spilled virtual register is used on the right hand side of a reduction, the value is retrieved from the spill area into a physical register. The physical register chosen must be of the class implied by the virtual register, but it need not be the same as the register in which the value was created. (See §8.10 for alternatives to unspilling.)

8.7.6. Register Spill Algorithm

When selecting a spillable register in the correct class, one must consider expected future use patterns for values in all physical registers in that class. Future uses are measured by a positive integer; a large value means the use is expected to be a long time in the future. There are three kinds of future uses. If a physical register is used in the pattern of the pending reduction, it has immediate future use. A value may have future uses by being included in the right hand side of a rule on a **reduce** action; the metric is the distance down the parsing stack to the first reference of the virtual register holding that value. Finally, the value may have future uses by being **shift**ed onto the parse stack; the metric is the number of symbols in the IR yet to be **shifted** before that value is encountered. Evaluated common sub expressions with future uses may have future uses both by **shifting** and by **reducing**. An attribute of each cse specifies when that cse will be used by **shifting**. (See §8.8.)

The register spill weighting functions use the future use metric, together with estimates for the short term probability of a **reduce** or of a **shift**. Since the next parsing actions, hence register uses, have locality of reference to the top of the parse stack or to the next few IR symbols, the register spilled should have the greatest future use metric, biased by the **shift** or **reduce** probabilities. If a double register is being spilled, the sum of the future use metrics for the constituent elements is used. The weight function makes reasonable assumptions that prefix expression trees are not too complicated and thus too deep, that binary operators predominate in use over unary operators, and that most instruction patterns contain many symbols. If the current Polish Prefix level is small (large absolute value), then there are assumed to be an abundance of binary operators over operands, so the probability of **shifting** must be high to correct this imbalance.

We now define characteristic values for both machine description grammars and for programs. By conventions used throughout this report, **P** is a machine description grammar for some target machine **M**. Let $|\mathbf{P}|$ be the number of syntactically distinct productions in **P**. If $g \in \mathbf{P}$, then $|g|$ is the length of the right hand side for g. Let $\mathbf{PR_{IR}}$ be a program represented in IR to be compiled for **M**. Assuming that $\mathbf{PR_{IR}}$ can be implemented on **M** using Glanville's code generation algorithm 4.1, let the bag **P** be the collection of productions in **P** used to derive $\mathbf{PR_{IR}}$. Clearly, the number of instructions to implement **PR** on **M** is $|\mathbf{PR}|$. The *static shift scale factor*, or *sssf*, and the *dynamic shift scale factor* or *dssf*, for program **P**, are defined as:

$$sssf = \sum_{\forall p \in \mathbf{P}} |p| \ / \ |\mathbf{P}| \qquad \textit{scale.10}$$

$$dssf = \sum_{\forall p \in \mathbf{PR}} |p| \ / \ |\mathbf{PR}| \qquad \textit{scale.15}$$

The *sssf* is the number of times the code generator parser expects to **shift** for each reduction, based on a static analysis of the grammar. The *dssf* is similar to the *sssf*, but is based on a dynamic analysis for a particular program (or history of programs) being compiled for **M**.

The computed weight functions currently used in **codegen** are scaled linearly. The functions have a *horizon* of 20. The horizon is a measure of locality of reference, either by performing a **shift** or a **reduce**. Any register with a future use metric greater than the horizon value will not be considered when deciding what to spill. *Stspill.10 - stspill.50*, in figure 8.1, are definitions for the probability and weight functions **codegen** uses to estimate which registers will be spilled without using any dynamic history. It is also possible to compute off line the actual probability functions for a given level and for a given program **PR**, as shown in *pspill.10 - pspill.20*, (figure 8.1).

$pr(x, l)$ = computed probability of x at level l *stspill.10*
$w(x, l, r_i)$ = weight of x at l for register r_i *stspill.15*

$pr(\textbf{reduce}, l) = \max(0, 0.05 \times l + 0.95)$ *stspill.20*
$w(\textbf{reduce}, l, r_i) = (\text{stack distance to } r_i) \times (1 - pr(\textbf{reduce}, l))$ *stspill.25*

$pr(\textbf{shift}, l) = 1 - pr(\textbf{reduce}, l)$ *stspill.30*
$w(\textbf{shift}, lr_i) = (\text{IR distance to } r_i) \times (1 - pr(\textbf{shift}, l))$ *stspill.35*
$= (\text{IR distance to } r_i) \times pr(\textbf{reduce}, l)$ *stspill.40*

$w(\text{use}, l, r_i) = w(\textbf{shift}, l, r_i) \times sssf + w(\textbf{reduce}, l, r_i)$ *stspill.45*
spill r_i such that $w(\text{use}, lr_i)$ is maximal *stspill.50*

$opr(x, l, \mathbf{P})$ = observed probability of x occuring at l for program **P** *pspill.10*

$nreduce(l, \mathbf{P})$ = number of reductions performed at l in program **P** *pspill.15*
$nshift(l, \mathbf{P})$ = number of shifts performed at l in program **P** *pspill.20*

$ssshift(l, \mathbf{P}) = nshift(l, \mathbf{P}) \times sssf^{-1}$ *pspill.25*
$dsshift(l, \mathbf{P}) = nshift(l, \mathbf{P}) \times dssf^{-1}$ *pspill.30*

$$opr_{static}(\textbf{reduce}, l, \mathbf{P}) = \frac{nreduce(l, \mathbf{P})}{(ssshift(l, \mathbf{P}) + nreduce(l, \mathbf{P}))}$$ *pspill.35*

$opr_{static}(\textbf{shift}, l, \mathbf{P}) = 1 - opr_{static}(\textbf{reduce}, l, \mathbf{P})$ *pspill.40*

$$opr_{dynamic}(\textbf{reduce}, l, \mathbf{P}) = \frac{nreduce(l, \mathbf{P})}{(dsshift(l, \mathbf{P}) + nreduce(l, \mathbf{P}))}$$ *pspill.45*

$opr_{dynamic}(\text{shift}, l, \mathbf{P}) = 1 - opr_{dynamic}(\textbf{reduce}, l, \mathbf{P})$ *pspill.50*

Figure 8.1: Static and per Program Spill Probability and Weight Functions

There has been little experience with the linear weighting functions, because they were implemented relatively recently, and because substantially better code, with less implementation overhead, can be produced using register preestimation and IR rewriting. To show that the weighting function is reasonable for complicated expressions that must spill, two artificially complicated IR programs were generated by the "D" compiler, and compiled by **codegen** for the PDP 11. ("D" is a subset of "C" with differing declaration syntax; see §9.2.) On our system, there are no real "C" programs containing complicated expressions, so we have no experience generating code for real world programs; these two artificial programs reflect worst case statistics.

The first program, the matrix multiply program taken from Glanville's dissertation (figure 5.6, page 94), was reworked to multiply matrices composed from complex numbers, using a straight forward algorithm. (Complex numbers were implemented as a third dimension on the matrices *A*, *B*, and *C* and the scalar *sum*). The observed probability distributions for this program are in Figure 8.2. A second FORTRAN program with very complicated expressions that Knuth describes as being

level	*Nreduce*	*Nshift*	*stred*	*dyred*
1	90	90	0.87	0.74
0	64	157	0.74	0.54
-1	50	188	0.65	0.44
-2	14	123	0.44	0.25
-3	0	56	0	0
-4	0	20	0	0
-5	0	3	0	0

KEY

sssf	static shift scale factor = 6.92
dssf	dynamic shift scale factor = 2.92
Nreduce	*nreduce*(l, P)
Nshift	*nshift*(l, P)
stred	opr_{static}(**reduce**, l, P)
dyred	$opr_{dynamic}$(**reduce**, l, P)

Figure 6.2: Compiling the Complex Matrix Multiply Program for the PDP 11

"rather horrible"[3] was rewritten in "D" and also compiled for the PDP11; the observed probability distributions are in Figure 6.3.

From the figures, it is apparent that the horizon of 20 is probably too much. Further, the assumptions about **shift** and **reduce** probabilities seem to be correct.

6.6. Common Sub Expressions

Computing and using common sub expressions (*cses*) fits easily into the grammar and the virtual register scheme. The TMDL grammar is extended by adding three operator symbols and, for each interesting register class, a production to evaluate and a production to use a *cse*. The additional operators and productions are not implemented by the target machine, but are caught and specially interpreted by **codegen**. The operator signifying cse evaluation, *compcse*, is a root level binary operator; the operator for cse use, *usecse*, is an interior (non root) unary operator. Glanville proposed that *compcse* (his O operator) be an interior operator; no generality is lost by the method **codegen** uses, which was chosen to minimize implementation overhead. Since the semantics of the implemented TMDL does not allow operators to have more than two operands, and since *compcse* logically has four operands, the interior binary operator *csedesc* is introduced solely to canonicalize more than two logical operands into a standard binary form.

All *cses* are initially evaluated into registers, although the register spill mechanism may later move the value into temporary storage. Since **codegen** is one pass, it can not consider all future uses of that *cse* and try to minimize execution time or space by fully, partially or not at all evaluating the *cse* into a register. For example,

[3]Knuth, D.E. *An Empirical Study of FORTRAN Programs*, Software Practice and Experience, 1:1, 1971. page 123.

level	*Nreduce*	*Nshift*	*stred*	*dyred*
1	74	74	0.87	0.74
0	120	35	0.96	0.91
-1	56	192	0.67	0.45
-2	65	163	0.73	0.53
-3	65	149	0.75	0.56
-4	53	149	0.71	0.50
-5	38	135	0.66	0.45
-6	18	97	0.56	0.35
-7	5	58	0.37	0.20
-8	0	24	0	0
-9	0	7	0	0
-10	0	1	0	0

KEY

sssf	static shift scale factor = 6.92
dssf	dynamic shift scale factor = 2.86
Nreduce	*nreduce*(*l*, **P**)
Nshift	*nshift*(*l*, **P**)
stred	opr_{static}(**reduce**, *l*, **P**)
dyred	$opr_{dynamic}$(**reduce**, *l*, **P**)

Figure 8.3: Compiling the Horrible FORTRAN Program for the PDP 11

when generating PDP11 code for this C expression:

$$a = (((b + c) + d) + c)$$

where all quantities are displacement addressable from r5 (automatic variables), and assuming no reordering performed on the associative and commutative operator + , one could produce the code alternatives shown in figure 8.4. Figure 8.4 gives the tuples (static instruction bytes, dynamic memory references). If code size were the only metric, then either *no cse* or *value cse* could be chosen. A *cse* generator, presumably machine independent, can not completely know the cost of the *cses* it proposes, because that cost depends intimately on the target machine. Ideally, a program like **codegen**, using exactly the same tables, would interact with the *cse* generator to ferret out worthwhile *cse* proposals before any code was emitted. This is a topic for future research.

The *cse* generator must supply some attributes that are machine independent. All *cse*'s have a unique identification. A *cse* has a non zero use count decremented each time the cse is used. Further, each evaluation or use of a *cse* can indicate where in the IR sequence, relative to the first symbol in the IR, that *cse* is used next. This next use value is used by the spill algorithm. As an internal check, the last use of a *cse* must have a negative next use. The general syntactic form for productions evaluating and using *cses* is hardwired into **codegen**. For the full PDP 11, there are three pairs of *cse* productions: double registers, floating registers and general registers. While two additional production pairs could be added for even and odd

```
 no cse            | value cse         | address cse
 (20, 15)          | (20, 14)          | (22, 16)
                   |                   |
mov  b(r5), r0     | mov  c(r5), r0    | mov  r5, r0
add  c(r5), r0     | mov  b(r5), r1    | add  $c, r0
add  d(r5), r0     | add  r0, r1       | mov  b(r5), r1
add  c(r5), r0     | add  d(r5), r1    | add  (r0), r1
mov  r0, a(r5)     | add  r0, r1       | add  d(r5), r1
                   | mov  r1, a(r5)    | add  (r0), r1
                   |                   | mov  r1, a(r5)
```

Figure 8.4: PDP11 Code Alternatives with Common Sub Expressions

registers, they are not needed since chain rules not emitting code convert freely between even and odd registers to general registers, and back again. The *cse* productions for the PDP11 general register class are:

```
•   → compcse csedesc k.1 csedesc k.2 k.3 r.1,
      "k.1 future uses, id k.2, k.3 next use, value r.1", magic = 1;
r.1 → usecse csedesc k.2 k.3,
      "id k.2, k.3 next use", magic = 2;
```

During a reduction, **codegen** treats a *cse* production like any other production, up to final register allocation and assembly code emission. Then, the **magic** production attribute springs a trap door keyed by the magic value, and **codegen** performs specialized semantic actions behind its own back. This trap door code checks the semantics of *cse* evaluation/use, performs any register allocation and cleans up. The trap door for *usecse* knows the register class of the *cse* specified, so can construct an appropriate left hand side. (All *usecse* productions are on the same reduce list because they have identical right hand sides, and are assumed to have left hand sides connected in the graph sense via chain rules.) The trap door code currently recognizes **magic** values 1 or 2; this mechanism can be extended to do other machine (in)dependent code generation in **codegen**'s implementation language. However, these extensions are hard to write, reduce **codegen**'s retargetability and may create maintenance and consistency problems between the IR generator, the machine grammar and **codegen**.

Glanville's matrix multiply algorithm was recompiled by hand into IR with three common sub expressions. (The address of the loop control variables *i*, *j* and *k* are recognized as *cses*.) The values of the variables were not treated as *cses* because they are used both as *rvals* and as *lvals*, and the grammar did not allow register variables as *lvals*. A "C" program taken from figure 5.6 (page 94) of Glanville's dissertation was written that forced the three available register variables to point to the loop control variables, in an effort to simulate the action of *cses* in the hand compiled program. Then, the IR was compiled into PDP11 code by **codegen**, and the C program was compiled by the Johnson PDP11 Portable C Compiler (PCC), the Ritchie (original) C compiler, and the Whitesmith C compiler. The code from these compilers is in figure 8.5.

While figure 8.5 is the matrix multiply code with three *cses*, figure 8.6 is code from the four compilers for the matrix multiply without *cses* that was taken from Glanville's dissertation.

Code generated by **codegen** with these three *cses* and with four *cses* (addresses of the variables *i*, *j*, *k* and *sum*), is in figure 8.7.

A number of comparisons can be made from these tests:

- The code produced by Henry's implementation using the 11 grammar can be compared against the code produced by Glanville's implementation using the *glan11* grammar taken from his dissertation (page 97). In particular, the 11 grammar recognizes multiply by two as an idiom.
- **Codegen**'s spill/unspill algorithm is exercised for three and four *cse* IR's.
- The spills/unspills performed by **codegen** for three *cse* IR can be compared to those done by the other compilers. There is no appreciable difference, except for the spill allocation strategy. (**Codegen** has not been fitted with a PDP11 spill area manager, so the spill code is hand generated.)
- One can observe the deteriorating and thrashing behavior of the code as more *cses* are recognized.
- In all "C" programs, variables were incremented using the construct

  variable = *variable* + 1

  instead of the construct

  variable++

  The Ritchie, Johnson and Whitesmith compilers do not recognize the first case of increment (where *variable* is a scalar) as a memory increment, but they do recognize the second case (again, where *variable* is a scalar) as a memory increment. On the other hand, **codegen** recognizes the first case efficiently. The TMDL grammar for the PDP11 could easily be extended to incorporate a new postfix operator, "++".

8.9. Register Classes Revisited

At any time, a physical register is associated with only one register class. Changing this association is done by a chain rule reduction, that usually does not emit any code. The goal partially or fully achieved by the chain reduction is to get the contents of the register into the "right" place, in preparation for emitting code implementing an operator requiring specially located or classified operands. This is where the power of a Polish Prefix intermediate form and the parser generator become apparent. Upon shifting over an operator symbol requiring operands in special registers, the parser knows that the operand(s) must be in a special register class. Assuming a uniform grammar, chain rules (and ordinary rules) eventually load the operand registers. However, reduction by chain rules is time expensive for the code generator, and code resulting from a poorly chosen set of chain rules may be very bad, especially if double registers are involved. The more register classes, the more chain rules. The TMDL writer should augment the grammar with just enough chain rules, knowing the structure of the operands for restricted operators. Chain rules to change register classes must be semantically ' = ' restricted with the same

Henry's 11

```
        mov   r5,r0
        add   $I,r0
        mov   r5,r1
        add   $J,r1
        mov   r5,r2
        add   $K,r2
        clr   (r0)
        jbr   L2
L1:     clr   (r1)
        jbr   L4
L3:     clr   SUM(r5)
        clr   (r2)
        jbr   L6
L5:     mov   $2*M,r3
        mul   (r0),r3
        add   (r2),r3
        asl   r3
        add   A(r5),r3
        mov   $2*M,r4
        mov   r1,spill_0
        mov   r4,r1
        mul   (r2),r1
        mov   spill_0,r4
        add   (r4),r1
        asl   r1
        add   B(r5),r1
        mov   (r1),r1
        mul   (r3),r1
        add   r1,SUM(r5)
        inc   (r2)
L6:     cmp   (r2),$M
        jlt   L5
        mov   $2*M,r1
        mul   (r0),r1
        add   (r4),r1
        asl   r1
        add   C(r5),r1
        mov   SUM(r5),(r1)
        inc   (r4)
L4:     cmp   (r4),$M
        jlt   L3
        inc   (r0)
L2:     cmp   (r0),$M
        jlt   L1
```

Johnson

```
        mov   r5,r0
        sub   $-I,r0
        mov   r0,r4
        mov   r5,r0
        sub   $-J,r0
        mov   r0,r3
        mov   r5,r0
        sub   $-K,r0
        mov   r0,r2
        clr   (r4)
L17:    cmp   (r4),$M
        jgt   L16
        clr   (r3)
L20:    cmp   (r3),$M
        jgt   L19
        clr   SUM(r5)
        clr   (r2)
L23:    cmp   (r2),$M
        jgt   L22
        mov   (r2),r1
        mul   $2*M,r1
        add   B(r5),r1
        mov   (r3),r0
        asl   r0
        add   r0,r1
        mov   (r1),-18(r5)
        mov   (r4),r1
        mul   $2*M,r1
        add   A(r5),r1
        mov   (r2),r0
        asl   r0
        add   r0,r1
        mov   (r1),r1
        mul   -18(r5),r1
        add   SUM(r5),r1
        mov   r1,SUM(r5)
L21:    mov   (r2),r0
        inc   r0
        mov   r0,(r2)
        jbr   L23
L22:    mov   (r4),r1
        mul   $2*M,r1
        add   C(r5),r1
        mov   (r3),r0
        asl   r0
        add   r0,r1
        mov   SUM(r5),(r1)
L18:    mov   (r3),r0
        inc   r0
        mov   r0,(r3)
        jbr   L20
L19:    mov   (r4),r0
        inc   r0
        mov   r0,(r4)
        jbr   L17
L16:
```

Ritchie

```
        mov   r5,r4
        add   $I,r4
        mov   r5,r3
        add   $J,r3
        mov   r5,r2
        add   $K,r2
        clr   (r4)
L5:     cmp   $M,(r4)
        jlt   L6
        clr   (r3)
L8:     cmp   $M,(r3)
        jlt   L9
        clr   SUM(r5)
        clr   (r2)
L11:    cmp   $M,(r2)
        jlt   L12
        mov   (r2),r1
        mul   $2*M,r1
        add   (r3),r1
        asl   r1
        add   B(r5),r1
        mov   (r1),-(sp)
        mov   (r4),r1
        mul   $2*M,r1
        add   (r2),r1
        asl   r1
        add   A(r5),r1
        mov   (r1),r1
        mul   (sp)+,r1
        add   SUM(r5),r1
        mov   r1,SUM(r5)
L13:    mov   (r2),r0
        inc   r0
        mov   r0,(r2)
        jbr   L11
L12:    mov   (r4),r1
        mul   $2*M,r1
        add   (r3),r1
        asl   r1
        add   C(r5),r1
        mov   SUM(r5),(r1)
L10:    mov   (r3),r0
        inc   r0
        mov   r0,(r3)
        jbr   L8
L9:     mov   (r4),r0
        inc   r0
        mov   r0,(r4)
        jbr   L5
L6:
```

Whitesmith

```
        mov   r5,r0
        add   $I,r0
        mov   r0,r4
        mov   r5,r0
        add   $J,r0
        mov   r0,r2
        mov   r5,r0
        add   $K,r0
        mov   r0,r3
        clr   (r4)
L1:     cmp   (r4),$M-1
        bgt   L3
        clr   (r2)
L11:    cmp   (r2),$M-1
        bgt   L5
        clr   SUM(r5)
        clr   (r3)
L12:    cmp   (r3),$M-1
        bgt   L32
        mov   (r3),r1
        mul   $2*M,r1
        add   B(r5),r1
        mov   (r2),r0
        asl   r0
        add   r0,r1
        mov   (r1),(sp)
        mov   (r4),r1
        mul   $2*M,r1
        add   A(r5),r1
        mov   (r3),r0
        asl   r0
        add   r0,r1
        mov   (r1),r1
        mul   (sp),r1
        add   SUM(r5),r1
        mov   r1,SUM(r5)
        mov   (r3),r0
        inc   r0
        mov   r0,(r3)
        br    L12
L5:     mov   (r4),r0
        inc   r0
        mov   r0,(r4)
        br    L1
L32:    mov   (r4),r1
        mul   $2*M,r1
        add   C(r5),r1
        mov   (r2),r0
        asl   r0
        add   r0,r1
        mov   SUM(r5),(r1)
        mov   (r2),r0
        inc   r0
        mov   r0,(r2)
        br    L11
L3:
```

Figure 8.5: PDP11 Matrix Multiply Code with Three Common Sub Expressions

Henry's 11

```
     clr  I(r5)
     jbr  L2
L1:  clr  J(r5)
     jbr  L4
L3:  clr  SUM(r5)
     clr  K(r5)
     jbr  L6
L5:  mov  $2*M,r0
     mov  r0,r1
     mul  I(r5),r1
     add  K(r5),r1
     asl  r1
     add  A(r5),r1
     mov  $2*M,r0
     mov  r0,r3
     mul  K(r5),r3
     add  J(r5),r3
     asl  r3
     add  B(r5),r3
     mov  (r3),r0
     mov  r0,r3
     mul  (r1),r3
     add  r3,SUM(r5)
     inc  K(r5)
L6:  cmp  K(r5),$M
     jlt  L5
     mov  $2*M,r0
     mov  r0,r1
     mul  I(r5),r1
     add  J(r5),r1
     asl  r1
     add  C(r5),r1
     mov  SUM(r5),(r1)
     inc  J(r5)
L4:  cmp  J(r5),$M
     jlt  L3
     inc  I(r5)
L2:  cmp  I(r5),$M
     jlt  L1
```

Johnson

```
     clr  I(r5)
L29: cmp  I(r5),$M-1
     jgt  L28
     clr  J(r5)
L32: cmp  J(r5),$M-1
     jgt  L31
     clr  SUM(r5)
     clr  K(r5)
L35: cmp  K(r5),$M-1
     jgt  L34
     mov  K(r5),r1
     mul  $2*M,r1
     add  B(r5),r1
     mov  J(r5),r0
     asl  r0
     add  r0,r1
     mov  (r1),-18(r5)
     mov  I(r5),r1
     mul  $2*M,r1
     add  A(r5),r1
     mov  K(r5),r0
     asl  r0
     add  r0,r1
     mov  (r1),r1
     mul  -18(r5),r1
     add  SUM(r5),r1
     mov  r1,SUM(r5)
L33: mov  K(r5),r0
     inc  r0
     mov  r0,K(r5)
     jbr  L35
L34: mov  I(r5),r1
     mul  $2*M,r1
     add  C(r5),r1
     mov  J(r5),r0
     asl  r0
     add  r0,r1
     mov  SUM(r5),(r1)
L30: mov  J(r5),r0
     inc  r0
     mov  r0,J(r5)
     jbr  L32
L31: mov  I(r5),r0
     inc  r0
     mov  r0,I(r5)
     jbr  L29
L28:
```

Ritchie

```
     clr  I(r5)
L18: cmp  $M-1,I(r5)
     jlt  L19
     clr  J(r5)
L21: cmp  $M-1,J(r5)
     jlt  L22
     clr  SUM(r5)
     clr  K(r5)
L24: cmp  $M-1,K(r5)
     jlt  L25
     mov  I(r5),r1
     mul  $2*M,r1
     add  K(r5),r1
     asl  r1
     add  A(r5),r1
     mov  (r1),r1
     mov  K(r5),r3
     mul  $2*M,r3
     add  J(r5),r3
     asl  r3
     add  B(r5),r3
     mul  (r3),r1
     add  SUM(r5),r1
     mov  r1,SUM(r5)
L26: mov  K(r5),r0
     inc  r0
     mov  r0,K(r5)
     jbr  L24
L25: mov  I(r5),r1
     mul  $2*M,r1
     add  J(r5),r1
     asl  r1
     add  C(r5),r1
     mov  SUM(r5),(r1)
L23: mov  J(r5),r0
     inc  r0
     mov  r0,J(r5)
     jbr  L21
L22: mov  I(r5),r0
     inc  r0
     mov  r0,I(r5)
     jbr  L18
L19:
```

Whitesmith

```
     clr  I(r5)
L13: cmp  I(r5),$M-1
     bgt  L33
     clr  J(r5)
L14: cmp  J(r5),$M-1
     bgt  L53
     clr  SUM(r5)
     clr  K(r5)
L15: cmp  K(r5),$M-1
     bgt  L35
     mov  I(r5),r1
     mul  $2*M,r1
     add  A(r5),r1
     mov  K(r5),r0
     asl  r0
     add  r0,r1
     mov  (r1),r1
     mov  K(r5),r3
     mul  $2*M,r3
     add  B(r5),r3
     mov  J(r5),r0
     asl  r0
     add  r0,r3
     mul  (r3),r1
     add  SUM(r5),r1
     mov  r1,SUM(r5)
     mov  K(r5),r0
     inc  r0
     mov  r0,K(r5)
     br   L15
L53: mov  I(r5),r0
     inc  r0
     mov  r0,I(r5)
     br   L13
L35: mov  I(r5),r1
     mul  $2*M,r1
     add  C(r5),r1
     mov  J(r5),r0
     asl  r0
     add  r0,r1
     mov  SUM(r5),(r1)
     mov  J(r5),r0
     inc  r0
     mov  r0,J(r5)
     br   L14
L33:
```

Figure 8.6: PDP11 Matrix Multiply Code Without Common Sub Expressions

Henry's 11, three cse

```
       mov   r5,r0
       add   $I,r0
       mov   r5,r1
       add   $J,r1
       mov   r5,r2
       add   $K,r2
       clr   (r0)
       jbr   L2
L1:    clr   (r1)
       jbr   L4
L3:    clr   SUM(r5)
       clr   (r2)
       jbr   L6
L5:    mov   $2*M,r3
       mul   (r0),r3
       add   (r2),r3
       asl   r3
       add   A(r5),r3
       mov   $2*M,r4
       mov   r1,spill_0
       mov   r4,r1
       mul   (r2),r1
       mov   spill_0,r4
       add   (r4),r1
       asl   r1
       add   B(r5),r1
       mov   (r1),r1
       mul   (r3),r1
       add   r1,SUM(r5)
       inc   (r2)
L6:    cmp   (r2),$M
       jlt   L5
       mov   $2*M,r1
       mul   (r0),r1
       add   (r4),r1
       asl   r1
       add   C(r5),r1
       mov   SUM(r5),(r1)
       inc   (r4)
L4:    cmp   (r4),$M
       jlt   L3
       inc   (r0)
L2:    cmp   (r0),$M
       jlt   L1
```

Henry's 11, four cse

```
       mov   r5,r0
       add   $I,r0
       mov   r5,r1
       add   $J,r1
       mov   r5,r2
       add   $SUM,r2
       mov   r5,r3
       add   $K,r3
       clr   (r0)
       jbr   L2
L1:    clr   (r1)
       jbr   L4
L3:    clr   (r2)
       clr   (r3)
       jbr   L6
L5:    mov   $2*M,r4
       mov   r1,spill_0
       mov   r4,r1
       mul   (r0),r1
       add   (r3),r1
       asl   r1
       add   A(r5),r1
       mov   $2*M,r4
       mov   r1,spill_1
       mov   r4,r1
       mul   (r3),r1
       mov   spill_0,r4
       add   (r4),r1
       asl   r1
       add   B(r5),r1
       mov   (r1),r1
       mov   r0,spill_2
       mov   spill_1,r0
       mul   (r0),r1
       add   r1,(r2)
       inc   (r3)
L6:    cmp   (r3),$M
       jlt   L5
       mov   spill_2,r0
       mov   $2*M,r1
       mul   (r0),r1
       add   (r4),r1
       asl   r1
       add   C(r5),r1
       mov   (r2),(r1)
       inc   (r4)
L4:    cmp   (r4),$M
       jlt   L3
       inc   (r0)
L2:    cmp   (r0),$M
       jlt   L1
```

Figure 8.7: PDP11 Matrix Multiply Code with Three and Four Common Sub Expressions

physical register on both sides of the chain rule.

Codegen allocates registers with a horizon constrained to the current production. While the parser's state implicitly contains allocation information, this may be difficult to extract without restructuring the grammar. In the current version of **codegen**, this information, available both on the stack and in the state, is *not* extracted and used. Consequently, choosing a destination register in the present may produce bad code in the future. An example comes from the matrix multiply example without *cse*'s, figure 8.5. Interesting productions in the TMDL grammar are:

```
o.1 → * ^ + k.1 r.2 o.1   mul  k.1(r.2), o.1
r.1 → k.1                 mov  $k.1, r.1
o=r1 → r=r1               ;    re classify r1 as odd
o.1 → r.1                 mov  r.1, o.1
```

codegen saw this fragment of IR:

```
··· * ^ + k.I r.r5 k.M ···
```

and produced this PDP11 code (an explanation follows):

```
mov  $M, r0
mov  r0, r1
mul  I(r5), r1
```

The grammar only knows how to get constants into general registers of class "r", so a load was emitted into the first free general register. Initially, all registers were free, and since registers are allocated in TMDL declaration order (r0, r1 ··· r4), r0 was chosen first. This was done even though the context of the "*" operator should have preferenced a register load into r1. Then, on the next reduction "o.1 → r.1" was performed, forcing an inter register shuffle to move the operand into the required odd destination target register.

One way to fix the problem of inadequate allocation lookahead is to rewrite the grammar, using the standard chain rule elimination algorithm by factoring chain rules into the grammar. (The chain rule elimination algorithm is applied to the grammar, and is not to be confused with the chain loop elimination algorithm used in **analysis**). The state of the parse now has encoded, among other things, the description of the most semantically restricted register that must be loaded.

For example, on the PDP 11, the general register class is the TMDL non terminal "r", classes "e" (r0 and r2), "o" (r1 and r3) and "t" (r4). The single production to move a displacement addressed quantity into a register would be duplicated into:

```
r.1 → ^ + k.1 r.2    original, "mov k.1(r.2), r.1"

o.1 → ^ + k.1 o.2
o.1 → ^ + k.1 e.1
o.1 → ^ + k.1 t.1

e.1 → ^ + k.1 e.2
e.1 → ^ + k.1 o.1
e.1 → ^ + k.1 t.1

t.1 → ^ + k.1 e.1
t.1 → ^ + k.1 o.1
t.1 → ^ + k.1 t.2
```

This "fix" increases the number of productions in the grammar by the square of the number of register classes. No real grammar has been modified this way, so the increase in table size and CGGWS execution time has not been measured.

8.10. Spills and Unspilling, Revisited

Codegen uses the register weighting algorithm from §8.7, and determines which physical register in which class is to be spilled. Currently, the register weighting algorithm does not check whether a physical register holds a *cse* that has already been spilled. (Respilling that register would cost nothing, as its value exists simultaneously in both a register and in the previous spill location.) **Codegen** relies on two machine specific routines written in **codegen**'s implementation language to do the actual spilling and unspilling. These routines are responsible for allocating slots in the spill area, and emitting code to do the spill (a move from a register) and do the unspill (a move to a register). (There is also a third machine specific routine that moves the contents of a register to another register; this is used to protect the contents of a dedicated register.) The **magic** trap door mechanism is not used here because the code is emitted internal to **codegen**, and has no ties to tokens in the IR.

There has been no machine specific spill and unspill written for **codegen**, since there are three bookkeeping difficulties.

(1) The spill area for most machines and languages must be allocated in cooperation with the variable storage area allocated by the IR generator. There is no mechanism in the IR so the IR generator can tell **codegen** where the spill manager can put spilled values.

(2) The spill manager can not use the run time stack in a pure last–in–first–out discipline, because spilling and unspilling is done with respect to register classes. Consider an example from the 8086, with an over abundance of register classes. Assume there are no *cses* and the stack contains references to values contained in "i" (index) and "b" (base) registers, as shown ("x" are attributes that do not concern us):

x x x i x x x b x x x

Assume that a request for an "b" register is made, and the one bound to the stack is spilled. Then, a request for an "i" register is made, again spilling the one bound to the stack. Now, a reduction is made encompassing the spilled "b", which lies beneath the spilled "i" value in the spill area.

(3) Registers need not be spilled out of the register set, as there may be a cheap inter register move to a register of another class. In the above example, the register of class "i" could have been cheaply spilled to a register in the "t" (temporary) class. However, even if the "t" class has a free member, there is no guarantee it will not be used immediately, necessitating a spill to memory anyway.

Spilled virtual registers are unspilled immediately before code for the pending reduction is emitted, so that the recursive application of a default list will not run out of registers. Spilled values are always unspilled into registers. One might produce better code automatically if the code to unspill were generated by the parser driven part of **codegen**, where fetching the spilled value might be encompassed as a side effect to another instruction. However, if the unspilled value is a *cse* with future uses in a particular and unknown environment, it may be better to unspill explicitly into a register.

There are two ways to use the body of **codegen** to perform unspills. Both methods macro expand a virtual register reference to a string of IR symbols that reference the spilled value. If the value to be unspilled is contained in the pending reduction, the pattern can be reparsed, expanding the spilled virtual register appropriately. Reparsing is necessary to recompute the state of the parser. Or, if a *cse* being shifted onto the stack is spilled, the IR scanner can expand the cse reference as a macro, instead of a virtual register reference. An alternative to macro expansion augments the TMDL vocabulary with a special non terminal representing a spilled value, and expands the grammar to handle, in as many instructions as possible, a value that is spilled. This method has advantages because it isolates spill policy into the TMDL. However, this method increases the size and complexity of the grammar, and will not avoid reparsing the pattern.

8.11. Codegen Retrospectus

The techniques used in **codegen** can generate high quality expression code if the implementing instructions use only registers from one class, and there are always enough registers in that class. While the machine description grammar can be inefficiently extended to differentiate between register classes and cure the first problem, the one pass, bottom up, left to right parse through the IR is simply not powerful enough to handle graciously any register spills and unspills that may be necessary. **Codegen**'s efforts to spill and unspill registers dynamically make it large, unreliable and potentially slow. Register preestimation and IR tree manipulation to avoid spills should be done with a table driven IR rewrite, preferably using the same tables and powerful parsing techniques as **codegen**. Register utilization can be determined from a skeleton code tree, and trees traversed and rewritten using ideas from the Sethi-Ullman code generation algorithm, but to handle register classes and their interactions, an algebra of vector Sethi-Ullman numbers must be defined. However, programs typically contain simple expressions. With these programs, the present version of **codegen** is competitive with other local–horizon only code generators, and is reasonably machine independent.

9. CGGWS Tools

9.1. MetaTMDL

9.1.1. TMDL Grammar Complexity

Simple target machine instructions with only one interesting destination perform one principal operation on a number of operands. The operand's address is specified using an *addressing mode*, specifying particular registers, constants or operators to use when computing the address of the operand. For truly orthogonal machines, all operand addressing modes can be used with all operators, although most machines impose addressability restrictions on certain operators. At the level of operator abstraction, the specific semantics of operand access are generally not important, although accessing the operand may perform more complicated arithmetic than the operator itself. All architecture reference manuals are organized on an operator (instruction) basis, enumerating the way operands may be addressed. In keeping with the abstraction, addressing mode semantics are separately specified.

TMDL lacks these levels of abstraction. The grammar is structured around a flat grammar, where each reduction reduces to either a non terminal corresponding to a destination register (or register pair) receiving a computed result, or a special, uninteresting, non terminal (λ) corresponding to a destination not in the register set. If a target machine operator has n operands, numbered $1 \cdots n$, and operand i can be addressed o_i different ways, then the TMDL grammar will have $\prod_{i=1}^{n} o_i$ productions for this operator. Alternatively there is one production for each instruction bit pattern, after register or constant semantics have been removed. Because the principal operator of an instruction is close to the root of the tree describing the instruction, flattening the tree places the principal operator in the left part of the modelling production's right hand side. An LR parser constructed with the methods currently used in **analysis** automatically merges only the states *before* the principal operator. Parser states recognizing the complicated operand trees to the principal operator can not be shared between other operators with identical operand trees, since the parser's state must encode the principal operator. (The present TMDL does not allow non terminals to group operators having identical operands.) Consequently, the parser has an enormous number of states. The number of states could be drastically reduced by factoring the grammar along the natural divisions of addressing modes. However, to achieve the reliability and efficiency offered by the flat grammar, it must be proved that a code generator constructed from a factored grammar will, for any valid IR input, automatically choose the best instruction sequence from alternative instruction sequences. The code generator built from the flat grammar does consider all possible instructions.

It is advantageous to use a flat grammar to recognize idiomatic instructions, assuming that the semantics of terminals and non terminals are specified using only one simple scalar value. There are currently two different idiom forms at the instruction/addressing mode level. *Binding* idioms require operands to have the same address. *Value* idioms require an operand attribute to have a value in a certain range. For example, the VAX's operator, **addl2**, is a binding idiom for **addl3**, while the **incl** instrution is a value idiom for **addl2**. With the flat grammar, the addressing mode structure of the operands is effectively lost, but if all the intra bundle "." bindings are met, the operands are equal. If the TMDL grammar were factored into addressing mode productions, the semantic attributes of an addressing mode non terminal would have to be defined. The definition must allow addressing modes to be compared as a whole for binding equality, and must allow value particulars to be tested. For most machines, however, the addressing mode semantics could not be encapsulated into a single scalar; these semantics vary from machine to machine. For example, a PDP11 addressing mode non terminal has four attributes: mode type, data type, register used, and constant used. The VAX has these four attributes, in addition to a second base register specifier. Extending TMDL and CGGWS to recognize factored grammars would require redesigning TMDL to know about arbitrary addressing mode semantics to recognize idioms.

9.1.2. CGGWS Organization for Factored Grammars

Assuming the theoretical feasibility, it would be a major undertaking to redesign TMDL and reimplement much of CGGWS to manipulate factored grammars. The reason is that the implementation of CGGWS makes deep, and in retrospect, erroneous assumptions about the register-only non terminals used in TMDL. Other more flexible parser generators, such as *yacc* can be used to write a machine specific code generator to test hypotheses about grammar factoring. However, these generators in their present state lack looping and blocking analysis, and can not construct default lists, although the latter two could conceivably be implemented by post processing the tables.

A TMDL was written for the micro instruction architecture of the PDP11. The register architecture contains Memory Buffer Registers (MBRs) and Memory Address Registers (MARs), (collectively called *multiplexor* registers). The MARs contain the address of an operand; the MBRs contain its value. Both are computed from an addressing mode. A set of patterns describing the micro instructions loading and storing these multiplexor registers was added to the PDP 11 grammar. The higher level (user visible) PDP11 instructions were redefined to work on the general register set and from the multiplexor registers containing the operands. Code generated from this machine description computes operands to real instructions in scattered places before the real instruction appears. If the intertwined microcode and high level code is reorganized by hand, the intermediate micro instructions subsumed by the real PDP11 instructions, and register re-allocation done, then normal PDP11 code is produced. After reorganization, the code for Glanville's Matrix Multiply (the one example the author tried), was identical to that produced with a flat grammar.

9.1.3. MetaTMDL

Because reimplementing CGGWS was a major proposition, the author was stuck with TMDL to experiment with and test other aspects CGGWS. After writing the *Z8000* grammar, and contemplating extensions to the *glan11* grammar, it became apparent that writing description grammars for regular architectures is a mechanical and tedious job, simply because there is so much duplication on a per instruction basis. The TMDL author must number and link the bundle semantics with the assembly string. Across productions of the grammar, one must ensure that no addressing modes are left out or incorrectly specified. The language metaTMDL was hastily designed and even more quickly implemented by **metatmdl** to incorporate addressing mode abstractions, to reduce the complexity of machine descriptions, and to number automatically and to construct TMDL instruction patterns. The metaTMDL compiler produces TMDL as its object code, so none of the modules **tmdl**, **analysis**, **merge**, **ccode**, and **codegen** had to be modified.

The metaTMDL compiler is a macro expansion preprocessor that evaluates the cross product of all addressing modes possible for an instruction. **Metatmdl** knows very little about the form of TMDL instructions, except that there is a production, an assembly string, and a cost. It knows there are "." bindings between the production and the string, and that the destination operand addressing tree appears in the production either on the very left of the right hand side, or as the left hand side. The brunt of the semantic checking is performed by **tmdl**, so errors that **tmdl** reports must be reflected back into the metaTMDL, a process that may take a number of iterations. The grammar is described in Appendix 3; what follows is an informal semantic and pragmatic description.

In this section, refer to figure 9.1, which is the metaTMDL description of the machine taken from Figure 3.1 (page 44) of Glanville's dissertation. (The metaTMDL description is longer than his dissertation example, not only because it commutes all operands to the "+" operator[1], but because the machine architecture uses a different set of operand addresses for sources than it does for destinations.) **Metatmdl** is designed with three observations in mind:

(1) Parts of a metaTMDL specification are for **tmdl** only. Consequently, metaTMDL files are organized like source files to *yacc*: text before an initial "%%" (in the first two columns) is copied out, text between that "%%" and another "%%" is processed by **metatmdl**, and text after the final "%%" is copied out. Text between the "%%" delimiters further delimited by "%{" and "%}" is copied out.

(2) Addressable operands described in TMDL can be cast into three different forms. The operand can be used as an lvalue, as an rvalue, or in the assembly string. It is assumed that the assembly string has the same form regardless of the operand's use as an lvalue or an rvalue.

(3) Principal operators take operands as sources and produce one destination operand. In the grammar rule, the operator's operands may appear in almost any order with respect to the operator; in the assembly string, their expansion can be surrounded by any text.

The metaTMDL **$modes** section defines the lval, rval and assembly semantic attributes for each addressing mode. In addition, the **cost** clause is the cost (by some scalar measurement) of using that instruction. If the target machine addressing mode contains a commutable operator, then the addressing mode should be repeated as many times as necessary to include all possibilities. Commuting the

[1] **Metatmdl** lacks a *commutative* attritube for binary operators, which would further reduce the size of metaTMDL descriptions and make them even easier to reduce.

```
$options statesets, tables;
$registers
        $allocatable {r0, r1, r2, r3, r4};
        $dedicated
$symbols
        $nonterminals
        r = r0, r1, r2, r3, r4, r5, r6, r7;
        $terminals
        k: -32768, 32767;
        +: binary;
        ^: unary;
        =: unary;
$instructions
%%
$modes                    /* begin of metaTMDL addressing mode definitions */
        reg =        assy:   "r.1",
                     lval:   "r.1 -> ",
                     rval:   "r.1", cost = 0;
        regindirect= assy:   "*r.1",
                     lval:   ". -> r.1",
                     rval:   "^ r.1", cost = 1;
        immediate=   assy:   "=k.1",
                     lval:   " ",
                     rval:   "k.1", cost = 1;
        absolute=    assy:   "k.1",
                     lval:   ". -> k.1",
                     rval:   "^ k.1", cost = 2;
        absoluteindirect=
                     assy:   "*k.1",
                     lval:   ". -> ^ k.1",
                     rval:   " ", cost = 1;
        offsetaddress1= assy: "=k.1,r.1",
                     lval:   " ",
                     rval:   "+ k.1 r.1", cost = 1;
        offsetaddress2= assy: "=k.1,r.1",
                     lval:   " ",
                     rval:   "+ r.1 k.1", cost = 1;
        offset1=     assy:   "k.1,r.1",
                     lval:   ". -> + k.1 r.1",
                     rval:   "^ + k.1 r.1", cost = 2;
        offset2=     assy:   "k.1,r.1",
                     lval:   ". -> + r.1 k.1",
                     rval:   "^ + r.1 k.1", cost = 2;
        offsetindirect1=
                     assy:   "*k.1,r.1",
                     lval:   ". -> ^ + k.1 r.1",
                     rval:   " ", cost = 3;
        offsetindirect2=
                     assy:   "*k.1,r.1",
                     lval:   ". -> ^ + r.1 k.1",
                     rval:   " ", cost = 3;
$endmodes;                /* end of metaTMDL addressing mode definitions */

$definitions              /* begin of metaTMDL addressing mode groupings */
        rvaladd =          ( offset1, offset2, absolute, reg );

        rvalload=          ( offsetaddress1, offsetaddress2,
                        offset1, offset2, absolute, regindirect, immediate );

        lvalstore=      ( offset1, offset2, offsetindirect1, offsetindirect2,
                        absolute, absoluteindirect, regindirect );

        register=       ( reg );
$enddefinitions; /* end of metaTMDL addressing mode groupings */

$instructions           /* begin of metaTMDL instructions */
%{
        /* this comment is copied out directly to the TMDL */
%}
        register := "+" rvaladd register   <"add" rvaladd "," register>, cost = 1;
        register := "+" register rvaladd   <"add" rvaladd "," register>, cost = 1;

        lvalstore := register               <"store" register "," lvalstore>, cost = 1;
        register := rvalload                <"load" register "," rvalload>, cost = 1;
$endinstructions;       /* end of metaTMDL instructions */

%%                      /* textual end of metaTMDL */

$end                    /* end of TMDL */
```

Figure 9.1: MetaTMDL for Sample Machine Description, from Glanville Figure 3.1

operands and the order of operators in addressing modes can exponentially increase the number of effective addressing modes. For example, to specify fully the VAX's displacement indexed addressing mode if no assumptions are made about the IR, 24 effective addressing modes must be enumerated. In addition, if the textual form of lvalues and rvalues in the assembly string differ, other effective addressing modes must be added.

The **$definitions** section defines *aggregates* of addressing modes. If a logical aggregate is twice used as an rvalue in an instruction pattern, the aggregate must be defined here twice, with different names but identical contents.

The **$instructions** section defines the instruction pattern. Each pattern consists of a prototype grammar rule constructor, a prototype assembly string constructor and an (optional) base cost. Both the rule and assembly string constructors are formed from *expanders*. An expander can be either a predefined aggregate name expanding to all addressing modes in the aggregate, or a quoted string which expands to itself. The grammar prototype consists of a left part separated from the right part with the reserved symbol ":=". The left part is the destination and can be only one expander; the right part is the pattern computing the source, and can be one or more expanders. An aggregate name may be duplicated across the ":=", but not within the right hand side because the resulting assembly string can be ambiguous. Grammar rules for this instruction are elaborated by evaluating the cross product of addressing modes from each unique aggregate named in the expander list, renumbering bundle qualifications as necessary to avoid conflict. The resulting grammar rule is formed from the lvalue and rvalue semantics (depending on the use in the prototype as destination or source) of the individual addressing modes. The cost of the rule is the sum of the costs of the prototype pattern's base cost, together with the cost of each addressing mode used by the expanders.

The assembly string prototype is delimited by < and >, and must contain the same aggregate names used in the rule pattern (although an aggregate name may be used more than once). The assembly string is constructed in parallel with the grammar rule, using the assembly string from the addressing mode definition.

The result of the "add" elaboration from figure 9.1 is in figure 9.2.

MetaTMDL lacks methods to commute automatically operators and operands in complicated addressing modes, although generating them can be done easily by hand with an editor. MetaTMDL also requires different names for addressing modes and aggregates, and only aggregate names can be used for elaborators, which taxes the metaTMDL writer's ability to generate names for complicated machines. MetaTMDL also rigidly adheres to a two level grammar; it is not possible to define addressing modes from less complicated addressing modes. **Metatmdl** is completely unforgiving for semantic errors, making it difficult to develop a machine description. However, for a user familiar with metaTMDL, the language is a reasonable way to enumerate instruction patterns.

9.2. IRgen

If writing TMDL is difficult, then writing IR for non trivial programs can be even more error prone. To debug **codegen** adequately, it needs to be run on different machines, with different IR test examples from a standard benchmark set. In addition, the spill and unspill algorithms would only become active in an interesting way if the IR test programs were horribly complicated at the expression level; such IR programs are difficult to write, and hard to modify. Consequently, the program **IRgen** was hastily written solely to generate complicated IR trees automatically from a language called "D" (for lack of a better name). The language "D" is a subset of the language "C", including all control structures and most operators, but not including structures, data types other than integer, and possesses a different declaration syntax from "C". IRgen works reasonably well.

```
r.1  →  + r.2 r.1            "add r.2 , r.1" cost = 1;
r.1  →  + ^ k.2 r.1          "add k.2 , r.1" cost = 3;
r.1  →  + ^ + r.2 k.2 r.1    "add k.2,r.2 , r.1" cost = 3;
r.1  →  + ^ + k.2 r.2 r.1    "add k.2,r.2 , r.1" cost = 3;
r.1  →  + r.1 r.2            "add r.2 , r.1" cost = 1;
r.1  →  + r.1 ^ k.2          "add k.2 , r.1" cost = 3;
r.1  →  + r.1 ^ + r.2 k.2    "add k.2,r.2 , r.1" cost = 3;
r.1  →  + r.1 ^ + k.2 r.2    "add k.2,r.2 , r.1" cost = 3;
```

Figure 9.2: MetaTMDL Produced ADD Instructions for Sample Machine Description

9.3. Metric

The tool **metric** was written to perform a number of experiments manipulating flat machine grammars. **Metric** interprets a specially formatted output from **tmdl** and determines (in much less time than **codegen** does) the number of states in the resulting parser. The number of states is useful for estimating the amount of time required by the programs in CGGWS to construct a code generator; if the initial estimate is too high when developing a code generator, then the grammar can be corrected. **Metric** is typically used in conjunction with **metatmdl**, when it is feared **metatmdl** will produce too many productions. **Metric** can also isolate common suffixes from the productions and determine the number of states the parser would have with the rewritten grammar. Isolating common suffixes does not affect the code produced by a parser, and typically halves the number of states in the parser. The rewritten grammar is not compatible with TMDL.

10. Conclusions and Future Directions for Research

Some of the original goals of the project have been met: we reimplemented Glanville's algorithms in the language "C" to provide a foundation for future experimentation with retargetable code generators. In the process, a number of errors and omissions in Glanville's algorithms were discovered. We gained practical experience writing machine description grammars for architectures more similar than different. These machine grammars were rich enough to uncover problems regarding register classes, multiple destination instructions and semantic attributes that could affect the quality of the code. Some of these problems have been adequately resolved, but fundamental efficiency and structural issues need more work. Machine descriptions for complicated machines like the VAX are too large to be useful. The present code generator produces poor quality code because it can not look ahead (or behind) to bias its register allocation or instruction selection. Future research will address these problems:

- Can the machine description grammar be factored so that the number of states in the code generator parser is drastically reduced? Initial experiments with a parser based code generator for the integer subset of the VAX are encouraging: the machine description grammar is factored into addressing modes, and less than 300 states. However, the heuristics used in writing the grammar and the support routines need to be formalized. We need to prove that the code generated from an SLR(1) or an LALR(1) parser constructed from an addressing mode factored grammar is equivalent to one constructed from the flat grammar.
- If the grammar is factored, the code generator may semantically block. How can the ideas suggesting default list construction in the flat grammar be used with a factored grammar?
- Can the tables produced by **analysis** (together with an table interpreter) form the machine dependent part of common sub expression analysis? How should the common sub expression proposer interact with **codegen**?
- How should TMDL be extended to model instructions with side effects, such as the PDP11 divide instruction, and the auto increment, auto decrement of a register found on both the PDP11 and the VAX?
- How powerful is TMDL for describing complicated IR operators and operands?

Our future research will address these, and other, issues with machine independent code generation using the Graham–Glanville techniques.

Page 78 is intentionally blank.

Acknowledgments

I am indebted to my research advisor, Susan L. Graham, for her guidance and patience while I developed, struggled with and got side tracked from the research project that partially culminates in this report. Acknowledgment, as always, goes to Bill Joy for his initial reimplementation of Glanville's Algorithms that were the foundation for this work. In addition, he has written the tools that make the environment what it is: *vi*, *csh* and a faster, hospitable kernel. John Foderaro and Kurt Shoens provided the teasing impetus to make measurable progress and produce something to show for the work. Ernie Co-VAX was always there giving a hand. R. Schulman, S. Feldman and P. Kessler read early drafts. Thanks also goes to my other non professional friends, parents and fiancee who provided additional and necessary emotional support.

References

[AhoUllman 77]	Aho, A.V. and Ullman, J.D. *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.
[Glanville 77]	Glanville, R.S. *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers*, Electronics Research Laboratory, University of California, Berkeley, UCB–CS–78–01, (December 1977).
[Glanville 78]	Glanville, R.S. and Graham, S.L. "A New Method for Compiler Code Generation", *Conf. Record Fifth ACM Symp. Principles of Programming Languages*, (January, 1978).
[Graham 78]	Graham, S.L. and Glanville, R.S. "The Use of a Machine Description for Compiler Code Generation", *Proc. Third Jerusalem Conference on Information Technology*, North Holland Publishing Co., (August 1978).
[Graham 80]	Graham, S.L. "Table–Driven Code Generation", *Computer*, **13:8**, (August, 1980), 25–33.
[Johnson]	Johnson, S.J. *A Tour Through the Portable C Compiler*, Bell Telephone Laboratories, undated.
[Wulf 75]	Wulf, William et. al. *The Design of an Optimizing Compiler*, Elsevier Computer Science Library, Elsevier North-Holland Inc., 1975.

Appendix 1: Execution Times

This appendix shows the execution times for various phases of CGGWS. Of particular interest is the high execution time required by the C compiler to compile the output of **ccode** into an object module to load with **codegen**. Note that the time required to construct default lists is exceptionally high, possibly reflecting inefficiences in the improved algorithm. Also, canonicalizing the IR, a trivial thing for **IRgen** to do, saves many parser states and CGGWS execution time.

Execution Times for CGGWS Modules (Glan 11 through Glan 360)					
metric	gl11	11	full11	nc11	gl360
cpp	.5	.5	.6	.5	.2
tmdlgen	.6	2.5	.41	2.7	.2
tmdl	1.7	7.8	13.9	13.2	1.5
# dlist	18	118	175	236	0
dlist	1.8	44.7	141	162	0
# state	197	782	1240	1406	193
CM	36K	137K	274K	267K	29K
analysis	12.7	194.1	668	551	
merge	4.1	20.0	42.0	35.7	3.7
ccode	4.7	17.1	30.5	34.6	3.6
cc	45.9	158	234	252	42.2
size	18K	61K	102K	113K	13K

Execution Times for CGGWS Modules (Z8000 through NAND)					
metric	Z8000	8086	3000	VAX	NAND
cpp	.4	.7	.2	.3	.1
tmdlgen	.2	.8	2.6	.1	.9
tmdl	3.0	9.1	4.7	16.0	.8
# dlist	29	124	42	426	42
dlist	1.2	48.9	11.6	369	.5
# state	316	936	672	1808	65
CM	55K	153K	96K	308K	8K
analysis	25.4	185	70.1	722	3.2
merge	5.8	22.2	12.9	56.9	1.0
ccode	6.8	24.5	10.8	43.4	2.0
cc	69.5	187	104	316	26.0
size	26K	76K	42K	147K	8K

Key

K	1000 bytes of storage
#	number
CM	Core Memory used in **analysis**
	otherwise, processing times in seconds

Appendix 2: TMDL, MetaTMDL and IR Recognized by CGGWS

Lexical Conventions Observed in *metatmdl*, *tmdl* and *codegen*

Source files to **metatmdl** and **tmdl** are free format. Spaces, tabs, line feeds, page feeds and comments are considered as separators. An arbitrary number of separators may appear between any two consecutive symbols in source files written in the languages metaTMDL, TMDL or IR, except they may not appear in identifiers, numbers of special symbols. At least one separator must appear between any consecutive pair of identifiers, numbers of keywords.

Comments in all three languages follow the C conventions, with an open delimiter of /* and a closing delimiter */. Comments may extend across lines.

In the grammars for metaTMDL and TMDL appearing in Appendices 3 and 4, respectively, keywords and single character symbols are in bold, (viz **comment**), non terminals in the grammar are in roman, (viz vdecllist), and other tokens having a lexical substructure are in italics (for example, *Identifier*). The keywords defining the major sections of a TMDL description (**comment**, **options**, **registers**, **allocatable**, **dedicated**, **symbols** and **instructions**) need not be preceeded by a "$", as is shown in Glanville's dissertation.

The lexical substructure of non reserved words and symbols is defined below using a modified regular expression notation recognized by *ed*, *ex*, or *vi*. In addition to the metacharacters "[", "]", "^", "-" and "*", we also use "{" and "}" to mean zero or one occurances of the bracketed character set, and "+" meaning one or more occurances of the preceding character.

```
  Identifier   [A-Za-z_][A-Za-z0-9_]*
  Identifier   ==[A-Za-z0-9_!@#%^&*-+|~?]*
  Identifier   <>[A-Za-z0-9_!@#%^&*-+|~?]*
  Identifier   !=[A-Za-z0-9_!@#%^&*-+|~?]*
  Identifier   [@#%^&*-+|~?][A-Za-z0-9_!@#%^&*-+|~?]*

EqIdentifier   =[A-Za-z_0-9]+                        Kludge
    EqNumber   =Number

      String   "[^"0*"

      Number   {-}[0-9]+                             decimal number
      Number   '[0-7][0-7][0-7]'                     character constant
      Number   {-}[0-9]+.[0-9]*{[eE]{+-}[0-9]+}      floating point number
      Number   {-}[0-9]*.[0-9]+{[eE]{+-}[0-9]+}      floating point number
```

Comments produced by the "C" preprocessor may be included in metaTMDL or TMDL source files. They serve only to change **metatmdl**'s and **tmdl**'s notion of the current source file and line number so that error messages produced by **tmdl** can be reflected back into metaTMDL. These comments are of the form:

# [0-9]+ "[^"\n]*"

It is recommended that source descriptions always be written in metaTMDL, using the macro definition and include file syntax recognized by the "C" preprocessor. These source descriptions should always be run through the "C" preprocessor, then through **metatmdl** and finally through **tmdl**. If none of the expansion facilities provided by **metatmdl** are used, then the "%%" delimiters should be left out of the description; **metatmdl** will pass the description unmodified.

Changes to Glanville's TMDL

A number of additions and changes have been made to Glanville's TMDL.

$comment A comment can be included in TMDL that is carried around with all machine description tables created from the TMDL source file. This is useful to tag listings and coders. The comment, along with date and time stamps is prepended to all listings produced by **tmdl**, **analysis**, **merge**, **ccode** and **codegen**. The comment must be inclosed in double quotes, and must appear at the head of the TMDL source file. For example,

$comment "This machine description is partially debugged";

does the trick.

$options Options control what the phases in CGGWS will print out as diagnostic information. Both options and comments are carried around with the machine description tables. An option name can be any identifier, and has effect only if a phase knows about it. Option names appear in italics in the manual pages (Appendix 6).

TMDL symbol names
Symbol names follow the semantics of identifiers, and are not constrained to single letters, as Glanville's implementation was.

Production Syntax
The syntax describing productions in the grammar has been extended to follow that used both in this report and Glanville's dissertation.

Double Register Qualifications
Double registers may be qualified on the left hand side of a production with the < · · · > syntax; see 3.2.

Production Qualifications
Productions can be qualified with the **cost** and **magic** keywords. The **cost** attribute is the scalar cost of the instruction, according to a cost function determined by the TMDL writer. Examples throughout this report use a cost function defined to be the number of memory references the instruction performs, including the instruction fetch. The **magic** attribute may be 1, implying a *cse* declaration, or 2, implying a *cse* use. See 8.8.

Appendix 3: Meta-TMDL Grammar

description:	modes definitions instructions ;
modes:	**modes** modelist **endmodes** ; | λ ;
modelist:	modedef | modelist modedef ;
modedef:	*Identifier* = mquallist modecostfactor ; ;
mquallist:	mqual | mquallist mqual ;
mqual:	qualtype : *String* ;
modecostfactor:	
	cost = *Number* | λ ;
qualtype:	**assy** | **lval** | **rval** ;
definitions:	**definitions** deflist1 **enddefinitions** ; | λ ;
deflist1:	defequal | deflist1 defequal ;
defequal:	*Identifier* = deflist ; ;
deflist:	**(** deflist2 **)** ;
deflist2:	*Identifier* | deflist2 *Identifier* ;
instructions:	
	insthead instlist **endinstructions** ; | λ ;
insthead:	**instructions**
instlist:	instdef | instlist instdef ;
instdef:	expander ::= expanderlist assydir instcostfactor ; ;
assydir:	< expanderlist > ;
expanderlist:	expander | expander expanderlist ;
expander:	*String* | *Identifier* ;
instcostfactor:	
	cost = *Number* | λ ;

Appendix 4: TMDL Grammar

tmdesc:	comment options registers symbols instructions **end** ;
comment:	**comment** *String* ; | λ ;
options:	optionlist ; | λ ;
optionlist:	**options** *Identifier* | optionlist , *Identifier* ;
registers:	**registers** allocatable
	| **registers** allocatable dedicated
	;
allocatable:	**allocatable** { reglist } ; ;
dedicated:	**dedicated** { reglist } ; ;
reglist:	*Identifier* | reglist , *Identifier* ;
symbols:	**symbols** variables terminals ;
variables:	**variables** vdecllist ;
vdecllist:	*Identifier* sep varlist ; | vdecllist *Identifier* sep varlist ; ;
sep:	= | : ;
varlist:	singlist | pairlist ;
singlist:	*Identifier* | singlist , *Identifier* ;
pairlist:	pair | pairlist , pair ;
pair:	< *Identifier* , *Identifier* > ;
terminals:	**terminals** tdecllist ;
tdecllist:	tdecl ; | tdecllist tdecl ; ;
tdecl:	oplist : *Number* , *Number*
	| oplist : *Number* . . *Number*
	| oplist : **unary**
	| oplist : **binary**
	| oplist **unary**
	| oplist **binary**
	;
oplist:	op | oplist , op ;
op:	*Identifier* | *Symbol* | = | < | > | : ;
instructions:	
	instructions instructionlist ;
instructionlist:	
	instruction ; | instructionlist instruction ; ;
instruction:	
	(qualoplist) rqualop semantics instquallist
	| (qualoplist) . semantics instquallist
	| rqualop Yields (qualoplist) semantics instquallist
	| rqualop Yields qualoplist semantics instquallist
	| . Yields (qualoplist) semantics instquallist
	| . Yields qualoplist semantics instquallist
	;
Yields:	::= | --> | -> ;
qualoplist:	qualop | qualoplist qualop ;
rqualop:	op < op . *Number* . *Number* >

```
               | qualop
               ;
qualop:        op
               | op . Number
               | op EqNumber
               | op EqIdentifier
               ;
semantics:     String | λ ;
instquallist:  instqual
               | instquallist instqual
               | instquallist , instqual
               | λ
               ;
instqual:        qualkind = Number
               | qualkind EqNumber
               | qualkind Number
               ;
qualkind:      cost | magic ;
```

Appendix 5: Machine Description Grammar for *11*, *nc11* and *full11*

```
$options
                FOOBAR
                /*,defaults*/
                /*,states
                /*,default*/
                ;

$registers
        $allocatable        {
                              r0 ,r1 ,r2 ,r3 ,r4
#ifdef FLOAT
                            , fr0 ,fr1 ,fr2 ,fr3
                            , fre4, fre5           /*good for inter FPP action*/
                            , fcc                  /*floating condition codes*/
#endif FLOAT
                            , cc
                            , stack1 ,stack2 ,stack3 ,stack4
                            };
        $dedicated
                            {r5, sp, pc};

$symbols
        $variables
                r=      r0, r1, r2, r3, r4, r5, sp, pc;     /*CPU registers*/
                c = cc;

#ifdef FLOAT
                fr=     fr0, fr1, fr2, fr3;                 /*FPP registers*/
                /*
                 *      fre are hidden FPP regs available only as reg immediates
                 */
                fre=    fre4, fre5;
                cf=     fcc;
#endif FLOAT
                e=      r0, r2;                             /*even registers*/
                o=      r1, r3;                             /*odd registers*/
                /*
                 *      d:      a true double length register, with
                 *              the low order word in the odd reg, and
                 *              the high order word in the even reg.
                 *      dprod:  a double register allocated just
                 *              to have two adjacent registers, but containing
                 *              just a single length result.
                 *              the even register contains only low order word
                 *              the odd register contains junk
                 */
                d=      <r0, r1>, <r2, r3>;
                dprod = <r0, r1>, <r2, r3>;

                stack = stack1, stack2, stack3, stack4;

$terminals
                k :     -32768, 32767;              /*16 bit integers*/
#ifdef FLOAT
                kf :    -1.0e+10, 1.0e+10;          /*32 bit floating point*/
#endif FLOAT
#ifdef BYTE
```

```
                    kb :     '\000', '\377';          /*8 bit bytes*/
#endif BYTE

                    + :binary;
                    - :binary;
                    * :binary;
                    / :binary;
                    % :binary;
                    m :unary;
#ifdef BYTE
                    +b :binary;
                    -b :binary;
                    mb :unary;
#endif BYTE
#ifdef FLOAT
                    +f :binary;
                    -f :binary;
                    mf :unary;
                    *f :binary;
                    /f :binary;
#endif FLOAT
             /*
              *     Jump operators
              */
                    j :      unary;
                    < :      binary;                   /*jump less than*/
                    <= :     binary;                   /*jump leq*/
                    == :     binary;                   /*jump equal*/
                    != :     binary;                   /*jump not equal*/
                    >= :     binary;                   /*jump geq*/
                    > :      binary;                   /*jump greater than*/

                    : :      unary;                    /*label operator*/
                    l :      0,32767;                  /*label numbers*/

             /*
              *     Condition code loads
              */
                    ? :      binary;
#ifdef FLOAT
                    ?f :     binary;
#endif
#ifdef BYTE
                    ?b :     binary;
#endif BYTE

             /*
              *     boolean algebra, on both integers and bytes
              */
                    | :      binary;  /* or */
                    & :      binary;  /* and */
                    ~ :      binary;  /* xor */
                    ! :      unary;   /* complement */
#ifdef BYTE
                    |b :     binary;
                    &b :     binary;
                    ~b :     binary;
                    !b :     unary;
#endif BYTE
```

```
            /*
             *      indirection and storing
             */
                    ^ :       unary;
                    gets :    binary;

#ifdef FLOAT
                    ^f :      unary;
                    getsf :   binary;

#endif FLOAT
#ifdef BYTE
                    ^b :      unary;
                    getsb :   binary;

#endif BYTE
            /*
             *      Common sub expressions
             */
                    compcse:        binary;         /*compute a cse*/
                    usecse :        unary;          /*use a cse*/
                    desc :          binary;         /*cse place holder for values*/
            /*
             *      Type coercions
             */
#ifdef FLOAT
                    wtof :          unary;          /*word to floating*/
                    ftow :          unary;          /*floating to word*/

#endif FLOAT
#ifdef BYTE
                    btow :          unary;          /*byte to word*/
                    wtob :          unary;          /*word to byte*/

#endif BYTE
            /*
             *      procedure call and return frobs, understood by
             *      IRgen
             */
                    stacksep:       binary;  /* argument evaluation */
                    force:          unary;   /* for a value to be returned */
                    entry:          unary;   /* procedure/func entry point */

                    funccall:       unary;   /* func call w/o args */
                    funccallargs:   binary;  /* func call with args */
                    funcreturn:     binary;  /* function return */

                    proccall:       unary;   /* proc call w/o args */
                    proccallargs:   binary;  /* proc call with args */
                    procreturn:     unary;   /* proc return */


$instructions
%%
          $modes
                    reg=      assy:     "r.1",
                              lval:     "r.1-->",
                              rval:     "r.1",          cost=0;
#ifdef BYTE
                    regb=     assy:     "r.1",
                              lval:     "r.1-->",
                              rval:     "r.1",          cost=0;
#endif BYTE
#ifdef FLOAT
                    regf=     assy:     "fr.1",
                              lval:     "fr.1-->",
                              rval:     "fr.1",         cost=0;
```

```
#endif FLOAT
          /*
           *    Indirect through a register
           */
                regind=        assy:     "*r.1",
                               lval:     ".--> gets r.1",
                               rval:     "^ r.1",          cost=1;
#ifdef BYTE
                regindb=       assy:     "*r.1",
                               lval:     ".--> getsb r.1",
                               rval:     "^b r.1",         cost=1;

#endif BYTE
#ifdef FLOAT
                regindf=       assy:     "*r.1",
                               lval:     ".--> getsf r.1",
                               rval:     "^f r.1",         cost=2;
#endif FLOAT
          /*
           *    indexed addressing
           */
                index1=        assy:     "k.1(r.1)",
                               lval:     ".--> gets + k.1 r.1",
                               rval:     "^ + k.1 r.1",    cost=2;
                index2=        assy:     "k.1(r.1)",
                               lval:     ".--> gets + r.1 k.1",
                               rval:     "^ + r.1 k.1",    cost=2;
#ifdef BYTE
                index1b=       assy:     "k.1(r.1)",
                               lval:     ".--> getsb + k.1 r.1",
                               rval:     "^b + r.1 k.1",   cost=2;
                index2b=       assy:     "k.1(r.1)",
                               lval:     ".--> getsb + r.1 k.1",
                               rval:     "^b + k.1 r.1",   cost=2;
#endif BYTE
#ifdef FLOAT
                index1f=       assy:     "k.1(r.1)",
                               lval:     ".--> getsf + k.1 r.1",
                               rval:     "^f + k.1 r.1",   cost=3;
                index2f=       assy:     "k.1(r.1)",
                               lval:     ".--> getsf + r.1 k.1",
                               rval:     "^f + r.1 k.1",   cost=3;
#endif FLOAT
          /*
           *    indirect addressing
           */
                indexind1=     assy:     "*k.1(r.1)",
                               lval:     ".--> gets ^ + k.1 r.1",
                               rval:     "^ ^ + k.1 r.1",            cost=3;
                indexind2=     assy:     "*k.1(r.1)",
                               lval:     ".--> gets ^ + r.1 k.1",
                               rval:     "^ ^ + r.1 k.1",            cost=3;
#ifdef BYTE
                indexind1b=    assy:"*k.1(r.1)",
                               lval:".--> getsb ^ + k.1 r.1",
                               rval:"^b ^ + k.1 r.1",      cost=3;
                indexind2b=    assy:"*k.1(r.1)",
                               lval:".--> getsb ^ + r.1 k.1",
                               rval:"^b ^ + r.1 k.1",      cost=3;
#endif BYTE
#ifdef FLOAT
                indexind1f=    assy:"*k.1(r.1)",
                               lval:".--> getsf ^ + k.1 r.1",
                               rval:"^f ^ + k.1 r.1",      cost=4;
```

```
                    indexind2f=         assy:"*k.1(r.1)",
                                        lval:".--> getsf ^ + r.1 k.1",
                                        rval:"^f ^ + r.1 k.1",      cost=4;
#endif FLOAT
          /*
           *        Constants
           */
                    const=              assy:     "#k.1",
                                        lval:     " ",
                                        rval:     "k.1",            cost=1;
#ifdef BYTE
                    constb=             assy:     "#kb.1",
                                        lval:     " ",
                                        rval:     "kb.1",           cost=1;
#endif BYTE
#ifdef FLOAT
                    constf=             assy:     "#kf.1",
                                        lval:     " ",
                                        rval:     "kf.1",           cost=2;
#endif FLOAT
          /*
           *        Specfic constants
           *
           *        k2..k32 are used for multiply and divide idioms
           */
                    k2=                 assy:     "1",
                                        lval:     " ",
                                        rval:     "k=2",            cost=0;
                    k4=                 assy:     "2",
                                        lval:     " ",
                                        rval:     "k=4",            cost=0;
                    k8=                 assy:     "3",
                                        lval:     " ",
                                        rval:     "k=8",            cost=0;
                    k16=                assy:     "4",
                                        lval:     " ",
                                        rval:     "k=16",           cost=0;
                    k32=                assy:     "5",
                                        lval:     " ",
                                        rval:     "k=32",           cost=0;
          /*
           *        Double registers
           *                 These are used for the CPU registers only, for
           *                        mul, div and ashc.
           *                 The logical double registers in the floating registers,
           *                 used for modd and modf are not considered..
           */
                    doublereg=          assy:     " ",
                                        lval:     "d.1 --> ",
                                        rval:     "d.1",            cost=0;

                    doublereg1=         assy:     "d.1.1",
                                        lval:     "d.1 -->",
                                        rval:     "d.1",            cost=0;
                    doublereg2=         assy:     "d.1.2",
                                        lval:     "d.1 -->",
                                        rval:     "d.1",            cost=0;

                    doublequoreg=       assy:     "d.1.1",
                                        lval:     "r<d .1 .1> -->",
                                        rval:     "d.1",            cost=0;
                    doublemodreg=       assy:     "d.1.2",
                                        lval:     "r<d .1 .2> -->",
                                rval:     "d.1",              cost=0;
                    /*
                     *      We must have two different registers for
                     *      multiply, enumerating all possible register
                     *      pairs in which multiply is done.
                     *      This is because a multiply takes a double
                     *      register of one flavor, and turns it into one of
                     *      another flavor. We can't have the direct production
                     *      d.1 --> * dprod.1 wordrval
                     *      because there is no linking between the left and right
                     *      hand sides (as there is in the instruction), and
                     *      the semantics of the two registers are different.
                     *      Therefore, we are stuck with enumeration, and use
                     *      of the '=' qualification method.
                     */
                    dprodreg0=      assy:     "r0",
                                    lval:     "d=r0 -->",
                                    rval:     "dprod=r0",     cost = 0;
                    dprodreg2=      assy:     "r2",
                                    lval:     "d=r2 -->",
                                    rval:     "dprod=r2",     cost = 0;

                    oddreg=         assy:     "o.1",
                                    lval:     "o.1 -->",
                                    rval:     "o.1",          cost=0;
                    evenreg=        assy:     "e.1",
                                    lval:     "e.1 -->",
                                    rval:     "e.1",          cost=0;
              /*
               *    pushes for actual params, and function returns
               */
                    stackslot=      assy:     "-(sp)",
                                    lval:     "stack.1 -->",
                                    rval:     "stack.1",      cost = 1;
                    funcretslot=    assy:     "r0",
                                    lval:     ". --> gets r=r0",
                                    rval:     "r=r0",         cost = 1;
              /*
               *    Condition codes
               */
                    cc=             assy:     " ",
                                    lval:     "c.1 -->",
                                    rval:     "c.1",          cost=0;
#ifdef FLOAT
                    ccf=            assy:     " ",
                                    lval:     "cf.1 -->",
                                    rval:     "cf.1",         cost=0;

#endif FLOAT
$endmodes;

$definitions
                    wordlval=       (reg
                                    ,regind
                                    ,index1
                                    ,indexind1
#ifdef NONCANON
                                    ,index2
                                    ,indexind2

#endif NONCANON
                                    );
                    wordrval =      (reg
                                    ,regind
```

```
                                        ,index1
                                        ,indexind1
                                        ,const
#ifdef NONCANON
                                        ,index2
                                        ,indexind2
#endif NONCANON
                                        );

                     wordrval1 =        (reg
                                        ,regind
                                        ,index1
                                        ,indexind1
                                        ,const
#ifdef NONCANON
                                        ,index2
                                        ,indexind2
#endif NONCANON
                                        );

#ifdef BYTE
                     bytelval=          (regb
                                        ,regindb
                                        ,index1b
                                        ,indexind1b
#ifdef NONCANON
                                        ,index2b
                                        ,indexind2b
#endif NONCANON
                                        );
                     byterval=          (regb
                                        ,regindb
                                        ,index1b
                                        ,indexind1b
                                        ,constb
#ifdef NONCANON
                                        ,index2b
                                        ,indexind2b
#endif NONCANON
                                        );
                     byterval1=         (regb
                                        ,regindb
                                        ,index1b
                                        ,indexind1b
                                        ,constb
#ifdef NONCANON
                                        ,index2b
                                        ,indexind2b
#endif NONCANON
                                        );

#endif BYTE
#ifdef FLOAT
                     floatlval=         (regf
                                        ,regindf
                                        ,index1f
                                        ,indexind1f
#ifdef NONCANON
                                        ,index2f
                                        ,indexind2f
#endif NONCANON
                                        );
                     floatrval=         (regf
                                                ,regindf
                                                ,index1f
                                                ,indexind1f
                                                ,constf
#ifdef NONCANON
                                                ,index2f
                                                ,indexind2f
#endif NONCANON
                                                );
#endif FLOAT


                          intreg=               (reg);
                          intreg1=              (reg);

                          dreg=                 (doublereg1);
                          oreg=                 (oddreg);
                          ereg=                 (evenreg);
                          dquoreg=              (doublequoreg);
                          dmodreg =             (doublemodreg);
                          dprodreg_0 =          (dprodreg0);
                          dprodreg_2 =          (dprodreg2);

                          stack =               (stackslot);
                          register0 =           (funcretslot);

                          power2=               (k2, k4, k8, k16, k32);
                          condcode=             (cc);
#ifdef FLOAT
                          floatreg=             (regf);
                          fcondcode=            (ccf);
#endif FLOAT

$enddefinitions;

$instructions
/*
 *        Loads, stores, and coercions
 */
          wordlval := wordrval                    <"mov" wordrval "," wordlval>;
          wordlval := "+" wordlval wordrval       <"add" wordrval "," wordlval>;
          wordlval := "+" wordrval wordlval       <"add" wordrval "," wordlval>;
          wordlval := "-" wordlval wordrval       <"sub" wordrval "," wordlval>;

#ifdef BYTE
          bytelval := byterval                    <"movb" byterval "," bytelval>;
          intreg := "btow" byterval               <"movb" byterval "," intreg>;
          bytelval := "wtob" intreg               <"movb" intreg "," bytelval>;
          intreg := "+b" intreg intreg1           <"add" intreg1 "," intreg>;
          intreg := "+b" intreg1 intreg           <"add" intreg1 "," intreg>;
          intreg := "-b" intreg intreg1           <"sub" intreg1 "," intreg>;
#endif BYTE

#ifdef FLOAT
          floatreg := floatrval                   <"ldf" floatrval "," floatreg>;
          floatlval := floatreg                   <"stf" floatreg "," floatlval>;
          floatreg := "wtof" wordrval             <"ldcif" wordrval "," floatreg>;
          wordlval := "ftow" floatreg             <"stcfi" floatreg "," wordlval>;
          floatreg := "+f" floatreg floatrval     <"addf" floatrval "," floatreg>;
          floatreg := "+f" floatrval floatreg     <"addf" floatrval "," floatreg>;
          floatreg := "-f" floatreg floatrval     <"subf" floatrval "," floatreg>;
          floatreg := "*f" floatreg floatrval     <"mulf" floatrval "," floatreg>;
```

```
        floatreg := "*f" floatrval floatreg          <"mulf" floatrval "," floatreg>;
        floatreg := "/f" floatreg floatrval          <"divf" floatrval "," floatreg>;
#endif FLOAT

%{

        /*
         *      Free transfers between single length registers
         */
        e=r0 --> r=r0                               ";r=r0 becomes e=r0";
        o=r1 --> r=r1                               ";r=r1 becomes o=r1";
        e=r2 --> r=r2                               ";r=r2 becomes e=r2";
        o=r3 --> r=r3                               ";r=r3 becomes o=r3";

        r=r0 --> e=r0                               ";e=r0 becomes r=r0";
        r=r1 --> o=r1                               ";o=r1 becomes r=r1";
        r=r2 --> e=r2                               ";e=r2 becomes r=r2";
        r=r3 --> o=r3                               ";o=r3 becomes r=r3";

        e.1 --> r.1                                 "mov r.1, e.1", cost = 1;
        o.1 --> r.1                                 "mov r.1, o.1", cost = 1;

        /*
         *      Formation of a double length result in preparation for
         *      a divide
         */
        d=r0 --> o=r1           "tst    r1\n\tsxt r0; div setup", cost=2;
        d=r0 --> e=r0           "mov    r0, r1\n\tsxt r0; div setup", cost=2;
        d=r2 --> o=r3           "tst    r3\n\tsxt r2; div setup" cost=2;
        d=r2 --> e=r2           "mov    r2, r3\n\tsxt r2; div setup" cost=2;
        /*
         *      Formation of a double length result in preparation for
         *      a multiply.
         */
        dprod=r0 --> o=r1       "mov    r1, r0; mul setup", cost=1;
        dprod=r0 --> e=r0       "; mul setup", cost = 0;
        dprod=r2 --> o=r3       "mov    r3, r2; mul setup", cost=1;
        dprod=r2 --> e=r2       "; mul setup", cost=0;
        /*
         *      Extracting the result of a multiply.
         *      Extracting the result of a divide is done automatically
         *      by the <..> qualification on divide and remainder.
         *      We don't use the same method for multiply, because
         *      a divide may follow immediately after the multiply,
         *      and there is no point is disassembling a register, and
         *      then immediately reassembling it.
         */
        o=r1 --> d=r0           "; discard HOW of d=r0", cost = 0;
        o=r3 --> d=r2           "; discard HOW of d=r2", cost = 0;
%}
        oreg := "*" oreg wordrval                   <"mul" wordrval "," oreg>;
        oreg := "*" wordrval oreg                   <"mul" wordrval "," oreg>;

        dprodreg_0 := "*" dprodreg_0 wordrval       <"mul" wordrval "," dprodreg_0>;
        dprodreg_0 := "*" wordrval dprodreg_0       <"mul" wordrval "," dprodreg_0>;
        dprodreg_2 := "*" dprodreg_2 wordrval       <"mul" wordrval "," dprodreg_2>;
        dprodreg_2 := "*" wordrval dprodreg_2       <"mul" wordrval "," dprodreg_2>;

        dquoreg := "/" dquoreg wordrval             <"div" wordrval "," dquoreg>;
        dmodreg := "%" dmodreg wordrval             <"div" wordrval "," dmodreg>;
```

```
/*
 *       negations
 */
         wordlval := "m" wordlval                     <"neg" wordlval>;
#ifdef BYTE
         bytelval := "mb" bytelval                    <"negb" bytelval>;
#endif BYTE
#ifdef FLOAT
         floatlval := "mf" floatlval                  <"negf" floatlval>;
#endif FLOAT
/*
 *       store idioms
 */
         wordlval := "k=0"                            <"clr" wordlval>;
#ifdef BYTE
         bytelval := "kb='\000'"                      <"clrb" bytelval>;
#endif BYTE
#ifdef FLOAT
         floatlval := "kf=0.0"                        <"clrf" floatlval>;
#endif FLOAT
/*
 *       Additive idioms
 */
         wordlval := "+" wordlval "k=1"               <"inc" wordlval>;
         wordlval := "+" "k=1" wordlval               <"inc" wordlval>;
         wordlval := "-" wordlval "k=1"               <"dec" wordlval>;
#ifdef BYTE
         bytelval := "+b" bytelval "kb='\001'"        <"incb" bytelval>;
         bytelval := "+b" "kb='\001'" bytelval        <"incb" bytelval>;
         bytelval := "-b" bytelval "kb='\001'"        <"decb" bytelval>;
#endif BYTE

/*
 *       Multiplicative idioms
 */
         wordlval := "*" wordlval "k=2"               <"asl" wordlval>;
         wordlval := "*" "k=2" wordlval               <"asl" wordlval>;
         intreg := "*" power2 intreg                  <"ash" intreg "," power2>;
         intreg := "*" intreg power2                  <"ash" intreg "," power2>;

         dreg := "*" power2 dreg                      <"ashc" dreg "," power2>;
         dreg := "*" dreg power2                      <"ashc" dreg "," power2>;

         wordlval := "&" intreg wordlval
                  <"com" intreg "\nbic" intreg "," wordlval>;
/*
 *       wordlval := "&" "!" wordrval wordlval        <"bic" wordrval "," wordlval>;
 *       wordlval := "&" wordlval "!" wordrval        <"bic" wordrval "," wordlval>;
 */
         wordlval := "|" wordlval wordrval            <"bis" wordrval "," wordlval>;
         wordlval := "|" wordrval wordlval            <"bis" wordrval "," wordlval>;
#ifdef BYTE
         bytelval := "|b" bytelval byterval           <"bisb" byterval "," bytelval>;
         bytelval := "|b" byterval bytelval           <"bisb" byterval "," bytelval>;
#endif BYTE

         intreg := "~" intreg wordrval                <"xor" wordrval "," intreg>;
         intreg := "~" wordrval intreg                <"xor" wordrval "," intreg>;
         wordlval := "!" wordlval                     <"com" wordlval>;
#ifdef BYTE
         intreg := "~b" intreg byterval               <"xor" byterval "," intreg>;
         intreg := "~b" byterval intreg               <"xor" byterval "," intreg>;
         bytelval := "!b" bytelval                    <"comb" bytelval>;
```

```
#endif BYTE

/*
 *       Comparisions
 */
         condcode := "?" wordrvall wordrval          <"cmp" wordrvall "," wordrval>;
         condcode := "?" wordrval "k=0"              <"tst" wordrval>;
         condcode := "?" "&" wordrval "k.6" "k=0"
                                                     <"bit" wordrval ", k.6">;
         condcode := "?" "&" "k.6" wordrval "k=0"
                                                     <"bit" wordrval ", k.6">;
#ifdef BYTE
         condcode := "?b" bytervall byterval         <"cmpb" bytervall "," byterval>;
         condcode := "?b" byterval "kb='\000'"       <"tstb" byterval>;
#endif BYTE
#ifdef FLOAT
         fcondcode := "?f" floatrval floatreg        <"cmpf" floatrval "," floatreg>;
         condcode := fcondcode                       <"cfcc">;
#endif FLOAT
%{
/*
 *       Branching
 */
         .--> : l.1                                  "\bLl.1:" cost = 0;
         .--> j l.1                                  "jbr Ll.1" cost = 1;
         .--> < l.1 c.1                              "jlt Ll.1" cost = 1;
         .--> <= l.1 c.1                             "jle Ll.1" cost = 1;
         .--> == l.1 c.1                             "jeq Ll.1" cost = 1;
         .--> != l.1 c.1                             "jne Ll.1" cost = 1;
         .--> >= l.1 c.1                             "jge Ll.1" cost = 1;
         .--> > l.1 c.1                              "jne Ll.1" cost = 1;

         .--> compcse desc k.1 desc k.2 k.3 r.1
                 ";compute reg (r.1) cse #k.2, k.1 next uses, first at k.3"
                 cost = 0 magic = 1;
         . --> compcse desc k.1 desc k.2 k.3 d.1
                 ";compute double (d.1.1) cse #k.2, k.1 next uses, first at k.3"
                 cost = 0 magic = 1;

         r.1 --> usecse desc k.2 k.3
                 ";use single (r.1) cse #k.2, next use at k.3"
                 cost = 0 magic = 2;
         d.1 --> usecse desc k.2 k.3
                 ";use double (d.1.1) cse #k.2, next use at k.3"
                 cost = 0 magic = 2;
#ifdef FLOAT
         fr.1 --> usecse desc k.2 k.3
                 ";use floating (fr.1) cse #k.2, next use at k.3"
                 cost = 0 magic = 2;
         . --> compcse desc k.1 desc k.2 k.3 fr.1
                 ";compute floating (fr.1) cse #k.2, k.1 next uses, first at k.3"
                 cost = 0 magic = 1;
#endif FLOAT
%}

         /*
          *       Procedure call and return frobs
          */
         stack := wordrval                           <"mov" wordrval "," stack>;
         stack := "stacksep" stack wordrval          <"mov" wordrval "," stack>;
         register0 := "force" wordrval               <"mov" wordrval "," register0>;

%{
        r=r0 --> funccall l.1               "jsr l.1\t;call function w/o args";
        r=r0 --> funccallargs l.1 stack.1
                                            "jsr l.1\t;call function with args";

        . --> proccall l.1                  "jsr l.1\t;call procedure w/o args";
        . --> proccallargs l.1 stack.1      "jsr l.1\t;call procedure with args";

        . --> procreturn l.1                "ret     ;procedure return from l.1";
        . --> funcreturn r=r0 l.1           "ret     ;function return from l.1";

        . --> entry l.1                     "\.align 2\n\t\.globl l.1\nl.1:";

%}

$endinstructions;

%%
$end
```

Appendix 6: UNIX[1]Manual Pages for CGGWS Modules

[1] UNIX is a trademark of Bell Telephone Laboratories.

NAME

CGGWS – Code Generator Generator's Work Station

DESCRIPTION

The Code Generator Generator's Work Station, or *CGGWS*, is a set of programs for building code generators from Target Machine Description Language (TMDL) [Glanville77], or from Meta Target Machine Description Language (metaTMDL) [Henry81]. Correct, although large, code generators for non trivial subsets of most computers can be constructed in a few hours with the programs and tools provided by *CGGWS*.

metatmdl *Metatmdl* is a preprocessor from metaTMDL to TMDL, and works like a restricted macro expander. It is useful for efficiently enumerating addressing modes that, if done by hand in TMDL, soon becomes tedious. Use of *metatmdl* is optional if one wants to write in TMDL.

tmdl *Tmdl* compiles TMDL into an internal form suitable for the code generator being constructed.

metric *Metric* is used in conjunction with *tmdl* to estimate quickly the number of states in the code generator parser that *analysis* will produce.

analysis *Analysis* constructs a parser from the machine grammar, using algorithms and techniques discussed in [Glanville77] and [Henry81]. The parser is represented as a set of tables.

merge *Merge* merges intermediate files together and compresses the tables describing the code generator.

ccode *Ccode* turns the tables for the code generator into initialized data structures in the language "C" that interface with the rest of the code generator.

IRgen *IRgen* compiles a language called "D" into Intermediate Representation (IR) [Glanville77] that *codegen* accepts as input to generate code from. "D" is based on "C".

codegen *Codegen* uses the tables created by *tmdl*, *analysis*, *merge* and *ccode* to generate code for the IR produced by *IRgen*, by hand, or by another compiler.

Codegen and *IRgen* are run once for each program being compiled. The other modules in the system are run once for each machine description, although in practice a number of iterations on the machine description to debug the IR interface are required before a satisfactory codegenerator can be built. Iteration time takes approximately 2000 seconds of VAX 11/780 CPU time for a machine description with the simplicity (or complexity, depending on how you look at it) of a PDP 11/70.

Assume you wish to use *CGGWS* to generate a code generator for your favorite computer, which is called "FOO". These steps will create a code generator for FOO from a metaTMDL machine description you supply in a file called *FOO.genmd*. (The suffixes and file names of sources are unimportant, although consistent nomenclature always helps.)

Machine Description editing phase:		
1	mkdir FOO.d	*Lots of files are produced; It is best to keep things separated*
2	cd FOO.d	*Go there*
3	vi FOO.gen.md	*Create your meta machine description*
4	metatmdl FOO.gen.md FOO.md	*Creates FOO.md complains about errors*
5	vi FOO.md	*Examine any errors... if errors go back to step 3*
6	tmdl FOO.md	*Analyze FOO.md...if errors find difficulty, reflect back to FOO.gen.md, and... go back to step 3*
7	tmdl −P FOO.md FOO.m.rules	*Create.. a rule summary for estimating states in file FOO.m.rules*
8	metric −E −d FOO.m.rules	*Count how many states... state metric appears on standard output If there are too many for this iteration, remove some implied productions by going back to step 3 or step 5*
9	tmdl −t FOO.md > FOO.rules -C"My FOO Machine"	*Get set to... Build the code generator "My FOO machine" is a comment prepended to all subsequent outputs Create a numbered listing of the rules in FOO.rules*

Table Construction Phase:

10	analysis −S −m 20 > analysis.run	*Build the parser* *This takes roughly 0.2 seconds for each state estimated by metric analysis.run contains statistics on the parser*
11	merge −p > merge.run	*Compress tables* *merge.run contains size statistics on the resulting code generator*
11a	merge −s −w440 −l2000 > FOO.tables	*Create a formatted listing of the parser's tables* *Do this only if you want to check visually for holes in the table indicative of blocking.*
12	ccode FOO.md.c	*Create an ...* *initialized "C" program into file FOO.md.c (which must end in ".c")*
13	cc −W −c −DREADONLY −DREADWRITE −Isourcedir.d FOO.md.c	*Create a ...* *compiled set of tables, ready to be loaded.* *sourcedir.d is the location of all source files implementing CGGWS Output in FOO.md.o*
14	cc FOO.md.o `cd sourcedir.d; nmake -f genmakefile` −o codegen	*Load the code generator modules with the tables.* *nmake on genmakefile enumerates the object modules implementing the table independent part of codegen.*

Codegen test phase:

15	vi test.D	*Create a test "D" program to test the code generator*
16	IRgen –F "fp" test.D > test.IR	*Compile into IR.. If bugs in test.D, go back to 15*
16a	vi test.IR	*Or, create an IR file by hand*
17	codegen test.IR > test.FOO.s	*Compile test.IR into machine code for FOO*
18	vi test.FOO.s	*Well.. examine the code for plausibility go back to step 1*
19	vprint FOO.rules analysis.run merge.run	*Print the rules and various statistics*
20	vpr -w -W FOO.tables	*Print the tables on a wide line printer (the Versatec at UCB)*

FILES

These file are hidden intermediate files between *tmdl*, *analysis* and *merge*. They are named with a standard naming convention, although they can be renamed; see the individual manual page documentation.

tmdl.out	Intermediate output of *tmdl*.
analysis.out1	Intermediate output of *analysis*.
analysis.out2	Intermediate output of *analysis*.
analysis.out3	Intermediate output of *analysis*.
merge.out	Intermediate output of *analysis*.

These files were generated in the *CGGWS* session above:

FOO.genmd	MetaTMDL source file describing FOO in terms of addressing modes.
FOO.md	TMDL source file from *metaTMDL* describing FOO in a more verbose way.
FOO.m.rules	*Metric* readable and digested machine grammar for FOO.
FOO.rules	This contains a listing of all symbols. For each rule, *FOO.rules* contains the rule pattern, the associated assembly string the restriction level, and the internal sequence number. The internal sequence number is referenced in *FOO.tables*.
analysis.run	Statistics regarding the parser and measurements of *analysis*'s performance.
merge.run	Statistics regarding the size of all tables composing the machine specific parts of the final code generator.
FOO.tables	This is a readable matrix of the parser's action and next table. It has been formatted to have up to 440/6 = 77 symbols across the columns, and up to 2000 rows. (If the parser is larger than that, the table will be printed in sectors.) This table is suitable for printing only on the widest printers.

FOO.md.c	"C" initialized data structures for the machine specific tables in the final code generator.
FOO.md.o	Object module form of the initialized data structures. In the example given, all tables reside in data space.
test.D	A test program written in "D" to test the code generator. This should be initially simple, and then more and more complicated.
test.IR	The IR compiled form of *test.D.*
test.FOO.s	The machine language compiled form of *test.D.*

These are the source files implementing *CGGWS*.

CGGWS include files:	
hash.h	symbol dictionary definitions
actiondefs.h	action definitions
seman.h	TMDL semantics
tabledef.h	the parsers action and next table
types.h	CGGWS wide type definitions
metatmdl:	
tgenmakefile	*nmake* format make file
tgenseman.h	internal model of metaTMDL
tgenmain.c	driver
tgen.y	parser for metaTMDL
tgenscan.c	metaTMDL character scanner
tgenhash.c	symbol dictionary
tgenTMDL.c	macro expansion
tgenseman.c	metaTMDL semantic processing
tmdl:	
tmdlmakefile	*nmake* format make file
tmdlseman.h	*tmdl* internal semantics
tmdlmain.c	driver
tmdl.y	yacc parser
tmdloptreg.c	process TMDL option and register declarations
tmdlsymbol.c	process TMDL symbol declarations
tmdlrules.c	process TMDL rule declarations
tmdllisting.c	list TMDL symbols and rules
tmdloutput.c	output intermediate file with TMDL semantics
metric:	
metric.h	definitions
metricmain.c	driver
metricinput.c	symbol and rule reading
metriccomp.c	metric computation and suffix isolation
metricoutput.c	rule output in yacc format
metricsubr.c	subroutines
parsedrive.c	lexical analyzer and driver for yacc parsers
analysis:	
analmakefile	*nmake* format make file
analreduce.h	reduce lists
analstates.h	state lists
itemsets.h	itemset manipulations
relations.h	bit matrices for symbol relations
analmain.c	driver
analmakerel.c	symbol relation constructor
analrelat.c	symbol relation manipulator
analitemset.c	itemset construction/manipulation
analgen.c	parser constructor driver
analdefault.c	default list construction
anallooping.c	loop detection and removal

analmreduce.c	reduce list construction
analpreduce.c	reduce list constructions print routines
analoutput.c	output intermediate file I/O

merge:

mrgmakefile	*nmake* format make file
mrgpack.h	*merge* table decomposition model
mrgseman.h	*merge* table decomposition model
mrgmain.c	driver
mrgsorttab.c	parser table sorting to find sub lists
mrgbuildlist.c	sub list construction from the parser tables
mrgoverlay.c	sub list suffix factoring and overlaying
mrgpack.c	sub list packing
mrgoutput.c	output routines

ccode:

ccodemakefile	*nmake* format make file
ccodemain.c	*ccode* driver
ccodetaout.c	*ccode* "C" file table output routines

IRgen:

IRgenmakefile	*nmake* format make file
IRgen.h	Semantics of "D" trees
IRtoks.h	token definitions
IRtoktab.h	token definitions for IRgen
IRtypes.h	data type definitions for IRgen
IRmain.c	driver
IRlex.c	lexical analyzer for "D"
IRparse.y	"D" parser in yacc
IRbuild.c	"D" tree constructors
IRcanon.c	"D" tree canonicalizer and factorizer
IRfold.c	constant folding and "D" tree compaction
IRgen.c	code generator and "D" tree walker
IRsubr.c	subroutines

codegen:

genmakefile	*nmake* format make file
genscan.h	IR scanner
genseman.h	register model
genmain.c	driver
genproto.c	prototypical machine description from *ccode*
genreginits.c	register model initialization
gencodegen.c	parser
genemit.c	reduce and default list application
genemitrule.c	register allocation interface
genconst.c	IR constant evaluation and range checking
genregallocs.c	register allocator
genspills.c	register spilling routines
gencse.c	common sub expression processing
genregprints.c	register allocation snapshot printing
genutils.c	utility routines

shared files:	
argproc.c	general UNIX process argc/argv argument parsing
initparser.c	*merge*, *ccode* parser table initialization
inittmdl.c	TMDL semantic initialization
lex.yy.c	*tmdl* and *analysis* lexical *analysis*
hash.c	symbol dictionary
actnextlook.c	*merge*, *codegen* parser table access routines
tableformat.c	*analysis* and *merge* parser table printer
iosubr.c	I/O subroutines
subr.c	general subroutines for printing and errors
malloc.c	memory allocator
utilities:	
nmake.c	source for *nmake*

89 Source Files comprising 24,600 lines of "C".

SEE ALSO

Additional manual pages may be found under:

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

Additional references may be found in:

[Glanville77] Glanville, R.S. *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers*, Electronics Research Laboratory, University of California, Berkeley, UCB–CS–78–01, (December 1977).

[Glanville78] Glanville, R.S. and Graham, S.L. "A New Method for Compiler Code Generation", *Conf. Record Fifth ACM Symp. Principles of Programming Languages*, (January, 1978).

[Graham78] Graham, S.L. and Glanville, R.S. "The Use of a Machine Description for Compiler Code Generation", *Proc. Third Jerusalem Conference on Information Technology*, North Holland Publishing Co., (August 1978).

[Graham80] Graham, S.L. "Table–Driven Code Generation", *Computer*, 13:8, (August, 1980), 25–33.

[Henry81] Henry, R.R. *The Code Generator Generator's Work Station: Reimplementation and Experimentation with the Graham-Glanville Machine Independent Algorithms for Code Generation* M.S. Project Report, Electronics Research Laboratory, University of California, Berkeley, (April, 1981).

AUTHOR

Robert Henry
Electronics Research Laboratory
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley
December 1978 - October 1980
Grant Numbers: NSF MCS74-07644-A04, MCS78-07291, MCS-8005144

NAME

metatmdl – Macro Expand meta-TMDL into TMDL

SYNOPSIS

metatmdl [infile [outfile]]

DESCRIPTION

The meta-TMDL is expanded into TMDL in this pass. The meta-TMDL comes from *infile* or the standard input if *infile* is absent or "=". The generated TMDL goes to *outfile* or the standard output if *outfile* is absent or "=".

Meta–TMDL contains descriptions of all the available addressing modes, an indication of which addressing modes are grouped together and can be used interchangeably, and a list of the instructions available on the target machine. These instructions are specified in terms of strings and the grouped addressing modes. Meta TMDL enumerates the cross product of all addressing modes implicitly specified in the grouped addressing modes given for each instruction. Using meta TMDL can substantially ease the burden of writing TMDL for highly regular machines, allowing one to make simple changes and deletions while experimenting with a machine description grammar.

FILES

By default metatmdl is invoked with these arguments:

metatmdl = =

SEE ALSO

CGGWS(henry)		
metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

DIAGNOSTICS

The meta TMDL analyzer attempts to pin point syntactic and semantic errors in the meta-TMDL, and give somewhat intelligent diagnostics. Error recovery is panic mode to the nearest major token, usually a ';'. Errors in meta-TMDL will produce incomplete TMDL, although the TMDL may be syntactically correct.

Some semantic errors are only warning errors. These include specifying an addressing mode with the number of "." or "=" qualified operands unequal amongst the lval, rval or assembly specifiers.

All error diagnostics are included as TMDL comments ("/*".."*/"), as well as printed on the standard error file.

Metatmdl understands the conventions for line numbering and file naming that the "C" preprocessor uses. All error messages are relative either to the beginning of the file, or to the file name/line number produced by the "C" preprocessor. The output of *metatmdl* to be input to *tmdl* generates file name/line number lines in the same format as the "C" preprocessor. With these mechanisms, it is relatively easy to pinpoint errors that *tmdl* may discover in preprocessed meta-TMDL.

BUGS

May core dump unexpectedly during error recovery.

NAME

tmdl – Syntactic and Semantic Analysis of TMDL

SYNOPSIS

tmdl [**–tP**] [**–C** *string*] [infile [outfile [intermediate files ...]]]

DESCRIPTION

This is the first phase of a system to produce machine independent code generators. Subsequent phases construct the code generator parser, compress tables and perform the code generation.

Tmdl performs syntactic and semantic analysis of a description of the target machine, written in Target Machine Description Language, (TMDL). The TMDL is presented to *tmdl* on *infile* or from the standard input if *infile* is absent or "=". The language accepted is almost identical to that described by Glanville[77], with extensions to improve the semantics of the code generation, and slight syntactic modifications to disambiguate some of Glanville's TMDL descriptions of double registers.

The output from this phase is used by all subsequent phases.

These options are available, and may also be set by including the names in italics in the **$options** section of TMDL:

–t Write to *outfile* or to the standard output if *outfile* is missing or is "=" the TMDL symbol table (*symtab*), which includes the grammatical symbols and rule patterns and their internally assigned sequence numbers. The output from this option is human readable.

–P Write to *outfile* or to the standard output if *outfile* is missing or is "=" a different version of the symbol table that the grammar postprocessor *metric* understands. The output from this option is barely human readable.

–C*string*

Include the *string* as a *comment* which will appear on all intermediate files produced from this run of *tmdl* and on all listings produced by future phases. The entire comment string is constructed from the name and date of the input file, the date *tmdl* was run on the input file, an optional comment from the **–C** option, and the comment string declared in TMDL by the **$comment** section. (This **$comment** section is an addition to Glanville's description of TMDL.)

Tmdl understands the lines inserted by the "C" preprocessor or *metatmdl* giving the file name and line number of the source file. All error messages are relative to these points.

FILES

tmdl.out	Intermediate machine description file used in subsequent phases.

By default, *tmdl* is invoked with these arguments:

tmdl = = *tmdl.out*

SEE ALSO

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

DIAGNOSTICS

Tmdl pinpoints syntactic and semantic errors with somewhat intelligent diagnostics. Error recovery is panic mode to the nearest major TMDL token, usually a semicolon.

Any errors forego generating the output file.

BUGS

May core dump for some kinds of input.

NAME

analysis – create a code generator parser

SYNOPSIS

analysis [**–sdS**] [**–M** *num*] [**–pirbLBDcI**] [intermediate files ...]

DESCRIPTION

Analysis is the code generator parser generator. It analyzes an encoded target machine description previously created by *tmdl*, and produces uncompressed tables describing the target machine and the code generator parser. *Analysis* uses SLR(1) parser construction techniques, examines the machine description for states that may block due to non uniformities in the target machine, removes potentially looping configurations from the parser caused by chain rules in the machine description grammar, and constructs default lists for semantically restricted rules. See Glanville[77] and Henry[80] for the algorithms.

Analysis catches interrupts. When interrupted, it will accept either another interrupt signal and terminate, or will accept another option argument string in the same form used when *analysis* is initially invoked. This way, the user can reenter a string of options and dynamically alter various options. To turn off an option, precede it with a lower case "n".

These options are the ones of most use:

–s As each *state* in the code generator is constructed, print out the shift/reduce information for that state. This option produces reams of output.

–d Print out the *default* lists constructed for semantically restricted rules.

–S Print out a readable *summary* of statistics about the resulting code generator. This includes the number of states, the number of items, the number and variety of rules, and information regarding the reduce and default lists.

–M *number*
Monitor (to the standard error output) what the state generator is doing every *number* states. This prints out the state number currently being constructed, and an estimate of the number of states yet to be constructed. This is useful for monitoring the progress of *analysis* when the resulting parser has many states.

These options are intended primarily to analyze the internal workings of *analysis*. The output they produce tends to be verbose and repetitive.

–p List the syntactically distinct rule *patterns* used to construct the parser.

–i List the *items* as they are constructed.

–r List the *relations* first, partial, final, dotprecedes and addlist.

–b List the states and symbols that will cause the code generator to *block*.

–L List the decisions the *loop* detection and removal algorithms make when finding and breaking loops in the grammar. Useful for debugging only.

–B Number states *breadthfirst*. (This is the default.)

–D Number states *depthfirst*. For both *depthfirst* and *breadthfirst* state numbering, non determinstic decisions in constructing the parser's finite state control graph are resolved in favor of traversing the "arc" in the control graph that is labeled with the smallest "alpha" number. Alpha order is defined to be the order in which variables and symbols are declared in the TMDL **$terminals** section.

−c List the *chainpath* relationship between variables causing potentially looping configurations in the parser.

−I Print out statistics on the *itemsets* generated, including their cardinality, and their folded representation.

FILES

tmdl.out	Output of *tmdl*.
analysis.out1	The first intermediate output file, containing the code generator parser's action, next and reduce list tables.
analysis.out2	The second intermediate output file, containing resource counters.
analysis.out3	The third intermediate output file, containing the default lists.

By default, *analysis* is run with these arguments:

analysis −B *tmdl.out analysis.out1 analysis.out2 analysis.out3*

SEE ALSO

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

BUGS

Some of the algorithms Glanville gives are not quite correct. See Henry[81] for revisions.

NAME

merge – Intermediate File Merging and Table Compression

SYNOPSIS

merge [**–pRC**] [**–w** *number*] [**–l** *number*] [**–PsOL**] [**–Sd**] [int. files ...]

DESCRIPTION

Merge logically merges the single intermediate file from *tmdl* and the three intermediate files from *analysis* into a single intermediate file. This single file is more suitable for use in the actual code generator, and is easier to save and manipulate.

Merge does not modify the logical representation of the parsing tables, but changes the physical representation from a matrix of numbers to lists of shared lists, physically compressing the tables to about 18% of their original size. The compression algorithm is basically heuristic in nature, sorting the table by both rows and columns to concentrate 'active' states and symbols into one corner, and then overlaying the common tails.

Merge prints out a nicely formatted table describing the parser's actions for each state and each symbol. For the 36" Versatec at UC Berkeley, the command for printing out the final sorted table is:

```
merge -s -w440 -l2000 | vpr -w -W
```

These options produce useful and terse output:

–p Print out table *packing* statistics. This option also prints out the byte sizes of the parser tables, symbol descriptors, default lists, reduce lists and rules constituting the final machine description as *codegen* requires it.

–R Print out *row* statistics, describing the common, shared sub lists for each row.

–C Print out *column* statistics, describing each column's activity.

These options are more verbose, printing out the logical and physical parser tables.

–P Print out the logical and physical unsorted table as the *parser* has constructed it.

–s Print out the logical and physical *sorted* table.

–O Print out the *overlayed* table. This is the physical representation of the logical table after recognizing common sub lists. This is probably of little use.

–L Print out the *logical* table from the sublist representation. This should be identical to the table produced under the **–s** option.

These options set the formatted size of all printed tables:

–w *number*
Print the table to occupy a *width* of *number* columns of output. The default is for the line printer: 132 columns.

–l *number*
Print the table to have a total *length* of *number* rows. The default is for the line printer: 60 rows.

These options are even more verbose. They are intended primarily for debugging.

–S Print out the *sublist* construction.

−d Print out the sublist *decomposition*. This provides information complementing the −S option.

FILES

tmdl.out	Intermediate machine description file from *tmdl*.
analysis.out1	First intermediate file from *analysis*, containing the parser's action, next and reduce list tables.
analysis.out2	Second intermediate file from *analysis*, containing resource counters.
analysis.out3	Third intermediate file from *analysis*, containing the default lists.
merge.out	Intermediate file from *merge*, containing the entire machine description.

By default, *merge* is run with these arguments:

```
    merge −w 132 −l 60
        tmdl.out merge.out analysis.out1 analysis.out2 analysis.out3
```

SEE ALSO

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

BUGS

The table sorting algorithm exhibits poor paging behavior.

NAME

ccode – Construct Initialized "C" Structures

SYNOPSIS

ccode [outfile [intermediate files ...]]

DESCRIPTION

Ccode takes the intermediate file from *merge* and constructs initialized data in the precise form that *codegen* requires it. *Ccode* produces a "C" program on *outfile* or the standard output if *outfile* is absent or "=". This "C" program is intended to be compiled and loaded with the other standard code generator modules, producing a code generator specific to the target machine described in a file created by a previous run of *merge*.

Having an initialized "C" program obviates the need for *codegen* to read and initialize its own data space, which, for large machines, takes a great deal of time. However, this convention is very sensitive to changes in structure definitions in *codegen*.

The output from *ccode* in *outfile* contains **#ifdef** macros that can be used to force read only tables into read only storage when *codegen* is loaded. The rest of the data is read/write.

FILES

merge.out	Intermediate output file from *merge*.

By default, *ccode* is run with these arguments:

ccode = *merge.out*

SEE ALSO

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

BUGS

If any structure or union definitions are changed, the order in which fields are written to *outfile* will have to be changed before the changes will be reflected into *codegen*.

Arrays of unions are initialized by renaming an array of structures using an "asm", as initialized unions are not defined in "C".

Changes to the allocation strategy in the "C" compiler may cause *ccode* to produce incorrectly initialized tables.

NAME

codegen – Table Driven Machine Independent Code Generator

SYNOPSIS

codegen [**-spPrtde**] [IR file [assembly code file]]

DESCRIPTION

Codegen parses the Intermediate Representation (IR) of the program being compiled, using a parser constructed from the application of the four previous phases *tmdl*, *analysis*, *merge* and *ccode*. *Codegen* allocates registers from those available on the target machine, spills registers to intermediate storage, and handles common sub expressions. The *IRfile* is a compiler generated Intermediate Representation of the source program, and must use the same tokens defined in the TMDL initially fed to *tmdl*.

Codegen is created by loading together a set of standard modules with a module containing initialized data fully describing the target machine. Retargeting *codegen* involves relinking, at the gain of quicker invocation each time the code generator is run.

Codegen recognizes these options:

–s Print out each IR token as it is *scanned* by the IR lexical analyzer.

–p Print out the *parser's* action for each IR token.

–P Print out the *parser's* stack for each IR token. Print out the allocation status of each register on the target machine before each reduction. Generally, this output is very verbose.

–r Print out the decisions the *register* allocator makes during a reduction. This information includes the register chosen to receive an intermediate expression result, the register chosen for a spill when there are no free registers, the nature of semantic restrictions between the left hand side and the right hand side of a instruction rule, and other generally useful information designed to make debugging easier.

–t Auxiliary *trace* flag, not currently used.

–d Auxiliary *debug* flag, not currently used.

–e Print out statistics about each *expression* after it is entirely reduced. These statistics include how many IR tokens formed the entire expression, how many reductions the parser took on that expression, the maximum depth of that expression tree and a summary of the statistics regarding shift and reduce probabilities for each polish prefix level in the expression.

–E Print out statistics for all expressions when an end of file in the IR file is encountered. These statistics are the sum of those gathered (and printed) by the **–e** option.

The syntax and semantics of the IR have been extended from what Glanville uses in his dissertation. Briefly, these extensions are:

— Wherever a numeric constant can occur, a symbolic constant can also occur. This allows the code generator to analyze examples similar to Glanville's Fig. 5.8, without translating the symbolic constants into magic numbers.

— Numeric constants may be either integers (base 10), character constants or floating point constants. The syntax for all three kinds of constants is identical to that used in "C".

— The IR may contain comments, enclosed in "/*" .. "*/" delimiters, as in "C". Text inside of these comments is discarded.

— The IR may contain *assembly* strings, enclosed in double quotes ('"') not extending over a line boundary. When these strings are encountered by the IR lexical analyzer, they are stripped of their double quote delimiters, and copied directly to the assembly language output file. The relative locations of the *assembly* strings and the instructions *codegen* produces are unpredictable unless the *assembly* strings occur between flattened expression trees. If this is the case, all code for the previous expression is emitted before the *assembly* string is copied out. Typically, *assembly* strings are used to pass through assembler directives, or when debugging *codegen* to echo the IR the codegenerator is working on.

FILES

IR file IR from the compiler. If the file name is absent, or given by a "=", the standard input is used.

Assembly code file
This file will receive the copied assembly strings and the final code. If the file name is absent, or given by a "=", the standard output is used.

By default, *codegen* is run with these arguments:

```
codegen = =
```

SEE ALSO

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

DIAGNOSTICS

IR that is invalid lexically, syntactically or semantically is diagnosed by a terse and specific error message before a core dump is taken.

Inconsistencies in the internal data structures (eg, attempting to free a register that is already free, and other crimes against reality) result in a readable dump of important internal data structures (including the parsing stack, register allocation snapshot, and extant common sub expressions) and eventually a core dump.

BUGS

Input lines may be no longer than 256 characters.

NAME

IRgen – Generate IR for the language "D", reminiscent of "C"

SYNOPSIS

IRgen [**–ydD**] [**–F***name*] [sourcefile [IRobjfile]]

DESCRIPTION

IRgen compiles into Intermediate Representation, or IR, a rather arbitrary and easy to implement tiny programming language reminiscent of "C". *IRgen* takes its input from *sourcefile* or if the argument is not present or "=", from the standard input. *IRgen* produces IR onto the output file *IRobjfile* or if that argument is not present or "=", to the standard output.

The language implemented is called "D", which is almost a subset of the programming language "C". IR is the standard intermediate form between a language specific compiler and *codegen*, a machine independent code generator. *IRgen* is intended to easy the difficulty of writing complicated IR test cases for *codegen* and to compare the output of the various "C" compilers with the output of *codegen*. Unfortunately, the declaration syntax for "D" is different from than that for "C", necessitating small changes in the sources, but the expressions accepted by "D" are a true subset of those accepted by "C".

IRgen was written as a quick and dirty tool to enable code generation for simple "C" like programs. A second reason for writing *IRgen* was to understand the techniques necessary to transform intermediate trees to reflect "C" semantics and to explore the realm of what was possible in TMDL semantics. "D" will become obsolete when the first pass of the portable "C" compiler is interfaced to a slightly modified *IRgen*.

"D" supports these operators and constructs with the same syntax (and hopefully semantics) as "C":

integers	floats	arrays	functions	procedures
if	if..else	do..while	while	for
break	continue	return	goto	labels
=	{ .. }	(..)	[..]	
+	–	*	/	%
&	|	^	|	
<	<=	=		
!=	>	>=		
&&	||	unary &	unary *	unary –
constants	actual params			

To whit, "D" does not support:

register	static	auto	extern	
pointers	structures	unions	function vars	
unsigned	char	short	long	double
?	:	,	++	––
<<	>>	op=		

In addition, all variables (global variables, formal parameters and local variables) are declared with an arcane syntax. For globals and formals,

var *name* <type> ;

does the trick. For locals,

var *name* **level** <leveldesc> <type> ;

does the trick. The *name* must be unique within the *sourcefile IRgen* is compiling; the symbol table is not flushed of formal declarations and local definitions on function declaration exit. The <type> specifies allocation and access semantics. *Ints* take up two bytes and *floats* take up four bytes. Arrays are allocated in row major order, with zero based indexing.

<type> ::= **int**
<type> ::= **float**
<type> ::= **array** [<integer>] **of** <type>

The level descriptor tells *IRgen* how to expand the access path to the particular variable. Globals are accessed using absolute addressing. Locals are accessed using indexing from the frame pointer. In addition, uplevel addressing may be simulated to any number of levels; in this case, the frame pointer is dereferenced an appropriate number of times before indexing is used.

<leveldesc> ::= **global**
<leveldesc> ::= **local**
<leveldesc> ::= <integer>

IRgen produces IR compatible with Glanville's dissertation PDP 11 TMDL, with extensions to handle the semantics of procedures. *IRgen* performs constant folding in the IR, canonicalizes the occurrence of constants so they are the left children of operators, prints out the IR in a nicely indented style, and comments the IR with the source program using the assembly string comment convention recognized by *codegen*.

IRgen recognizes these options:

−y Turn on *yacc* debugging when the "D" source is parsed.

−d Turn on *debugging*.

−D Turn on a more verbose *debugging*.

−F *name*
The name of the frame pointer is *name*.

FILES

IRgen is invoked by default with these arguments:

IRgen −F r5 = =

SEE ALSO

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

DIAGNOSTICS

Complains about invalid "D", and refuses to compile any more.

BUGS

The type propagation and interaction between floats and integers is flakey.

NAME

metric – Analyze and Measure a Postprocessed TMDL grammar

SYNOPSIS

metric [–MEDd] [–sSPF] [–Y yaccfile] [grammfile [sumfile]]

DESCRIPTION

Metric analyzes the **–P** postprocessor formatted and digested output *tmdl* produces from Target Machine Description Language. Analyzing the previously digested TMDL relieves *metric* from having to parse and understand TMDL, a job which *tmdl* is best suited for. This digested input contains an enumeration of all symbols defined and their polish prefix weight (one per line), and a list of all rules with their TMDL assembly string (one per line). The rules have been stripped of semantic qualifications. *Metric* takes its formatted input from *grammfile* or, if not given or "=", from the standard input. The summary output, which is enabled by setting various options, goes to the *sumfile* or, if not given or "=", to the standard output.

Metric computes the prefix and suffix *match metric* which is used to estimate the number of states a parser constructed from the TMDL grammar will have. *Metric* also finds common suffixes in the grammar that can be factored out, and creates a *yacc* formatted machine description grammar, with associated actions consisting of *printf*s of the TMDL assembly string. Finding and factoring common suffixes is useful for reducing the number of states in a parser. The yacc grammar can be used with *yacc* to test the effectiveness of a factored grammar, and compare the size of the tables *yacc* produces for both factored and unfactored grammars with the size of tables the *merge* phase produces.

These options control which metrics are computed and output to *sumfile*:

–M Print out the prefix and suffix match *metric*.

–E Print out the *estimate* of the number of states an LR parser (prefix match metric) and RL parser (postfix match metric) would have.

–D Print out the rules after they are sorted for common prefixes or common suffixes. Used for *debugging*.

–d Print out a count of the number of syntactically duplicated rules in the *grammfile*.

These options control how suffixes are isolated:

–S Print out *suffixes*.

–s Convert the machine description grammar in *grammfile* to use *suffixes*.

–P Find suffixes using the *principal* operator method outlined below. Exclusive from the **–F** option.

–F Find suffixes using the general suffix match metric. Exclusive from the **–P** option.

This option controls the how the *yacc* input file is generated.

–Y *name*
A *yacc* compatible file is to be created in *name*. The grammar produced depends on the settings of the **–P**, **–E** and **–s** options.

Metric works by finding syntactically common prefixes and suffixes in the grammar. The prefix match metric for a grammar production is the length of the longest prefix of the right hand side that is shared with another production.

Shared prefixes reduce the state count of an LR parser constructed from the grammar. The suffix match metric is similar, except it is for shared suffixes. Shared suffixes reduce the state count if common suffixes are factored out as separate productions, or if an LR parser is constructed from reversed productions in the grammar. A function of the prefix match metric over all productions will quickly and accurately estimate the number of states in the parser that *analysis* will produce for the TMDL.

Metric also finds common suffixes by considering the prefix weight of each grammar production. For machines with a reasonably uniform combination of operators and operands, a *principal operator* can be isolated out of the right hand side of each production. This principal operator reflects the principal semantics of the instruction on the target machine being modeled by the production. The prefix string composing the concatenation of the operands to the principal operator usually is shared among similar operators in a number of productions. Hence, these strings are good candidates for shared suffixes. The principal operators are not isolated into suffixes so the semantics of the modeled instruction are not factored out. The destination of the result of the principal operator is a prefix that is automatically shared by the LR parser construction.

FILES

Metric is invoked by default with these arguments that may be changed to redirect the input or output:

```
        metric -P = =
```

SEE ALSO

CGGWS(henry)

metatmdl(henry)	tmdl(henry)	analysis(henry)
merge(henry)	ccode(henry)	codegen(henry)
IRgen(henry)	metric(henry)	

DIAGNOSTICS

Complains if the *grammfile* is not in an understandable format.

NAME

nmake – Preprocessor for make

SYNOPSIS

nmake [**–#**] [**–?**] [**–f** makefile] [makeoptions]

DESCRIPTION

Nmake, with the help of *make*, helps to manage the construction of software systems composed of many programs. The dependencies and syntax *make* understands is not powerful enough to tersely specify dependencies between a large number of source files producing even more object files that in turn are shared between various programs. *Nmake* also obviates the need to use the baroque and occaisonaly undesired general dependency rules *make* uses.

Nmake's model of a make assumes that object modules are created from "C" source modules. Some object modules have special rules to make them by, and have names not at all related to the name of the source module. Interesting source modules can be printed out in a specific order, and all source modules should be briefly documented in the *makefile* describing what they contain.

Nmake expands the *makefile* (or if that is not given, *nmake* looks for one of *Makefile*, *makefile*, *nMakefile* or *nmakefile* in the current directory in that order) into a makefile that *make* understands. The general format of recognized makefiles has three sections, separated by two "%%" section boundarys, much as *lex* or *yacc* source files do. Any lines starting with a "#" in any of the three sections *make* comment delimiter are normally discarded.

Nmake passes out undisturbed any text preceding the first section boundary. Text between the first and second section boundarys is processed by *nmake* as the *dependency table*. At the end of the dependency table, a number of *make* macro definitions are defined from information gleaned from the dependency table. Text after the second section boundary is also stripped of comments and passed out. This text is the dependency information *make* will need to complete the make file, and should define macros that the dependencies *nmake* constructs may use. When the end of file is encountered, a list of *make* dependencies and actions in a standard format is constructed from the dependency table. *Nmake* then calls *make do the actual work.*

Nmake recognizes a dependency table with 4 tab or space separated fields. Each field in the table describes one source–object file pair in detail.

SourceFileName (Field 1)
: The *SourceFileName* can be anything, although files ending with ".h" or ".c" have special meanings. The *SourceFileName* may be preceded with a "!" or "@", which is stripped off, but effects the constructed dependencies.

ObjectFileName (Field 2)
: The *ObjectFileName* can be anything, although *ObjectFileNames* ending with ".o" (as expected) have special meaning. The *ObjectFileName* may be preceded with an "@" which is stripped off.

print order (Field 3)
: The print order is a string of digits, optionally preceded with an "@".

comment (Field 4)
: This field is ignored.

These *make* macros are defined from the information gleaned from the dependency table:

HDRS All *SourceFileNames* with ".h", not preceded by "@", sorted alphabetically.

CSRCS All *SourceFileNames* with ".c", not preceded by "@", sorted alphabetically.

UNKSRCS
All *SourceFileNames* not in HDRS and CSRCS and not preceded by "@", sorted alphabetically.

PRINTS All *SourceFileNames* whose printorder field is not preceded by an "@", sorted numerically by the printorder field.

OBJS All *ObjectFileNames* ending with ".o", not preceeed by "@", sorted alphabetically.

The dependencies *nmake* constructs have one simple form, with a possible addition to handle differing *SourceFileName* and *ObjectFileName* roots. For each *SourceFileName* ending with ".c" not preceded by a "!", having an *ObjectFileName* ending with ".o", the make directive

```
ObjectFileName : SourceFileName $(DEPEND)
        $(COMP) SourceFileName
```

is constructed. In addition, if the root of the *SourceFileName* differs from the root of *ObjectFileName*, then the third line:

```
        mv SourceFileRoot.o ObjectFileName
```

is added after the first two commands in the directive.

Nmake recognizes these options:

–# Pass through comment lines starting with "#".

–? Produce a *make* compatible makefile called MAKEFILE in the current directory. Do not run *make*.

–f *name*
Find the *nmake* compatible makefile in *name* instead of in the default locations *Makefile* or*makefile*.

All other arguments to *nmake* are passed to *make* undisturbed.

FILES

Makefile	First default make file searched for if **–f** *name* is not given.
makefile	Second default make file searched for if **–f** *name* is not given.
/tmp/RRHmakeXXXXXX	Expanded makefile in *make* format when *make* is to be run.
MAKEFILE	Expanded makefile in *make* format when debugging **–?** option is set and *make* is not run.

SEE ALSO

make(1)

```
PATH = ../glan11.d/
MDNAME = glan11.md
OPTIONS =
#
#         Makefile to construct individual codegenerators from
#         modules composing the prototypical code generator, yielding codegen
#
#         the modules contain all of the functions that implement
#         the codegenerator, but does not contained the initialized
#         data that defines the code generator for a particular machine.
#
#         The initialized data can fall into either the text segment (read
#         only), or the data segment (read/write).
#
#         This Makefile assumes that machine descriptions are in standard
#         places with respect to the location of this Makefile.
#
#         Flags appropriate to C compiling the machine description:
#
#         READONLY        Set to extract the read only initialized data,
#                         so one can change all data segments to
#                         text segments in the file md.s
#
#         READWRITE       Set to extract the read/write initialized data.
#
#
MD = $(PATH)$(MDNAME)
SOURCEDIR = /va/staff/henry/tmdl/sourcedir.d/
UTILPATH = /va/staff/henry/tmdl/
#
#         NOTE: the C compiler will create the .s file (from the -S option)
#         in the current! directory
#
RO_COMPILE = $(CC) -S -w -DREADONLY -I$(SOURCEDIR)
RW_COMPILE = $(CC) -c -w -DREADWRITE -I$(SOURCEDIR)
ALL_COMPILE = $(CC) -c -w -DREADONLY -DREADWRITE -I$(SOURCEDIR)

ROFIX = sh ./:rofix

CFLAGS = -g -p
LINK = $(CC) $(CFLAGS)

EXAMPLE1: $(MD).o genobjs
        $(LINK) `cat genobjs` $(MD).o -o $(PATH)codegen

genobjs:        codegen

codegen:
        nmake -f genmakefile buildobjs

$(MD).o:        $(MD).c
        $(ALL_COMPILE) $(MD).c
        mv $(MDNAME).o $(MD).o

$(MD).c:        $(PATH)merge.out
        $(UTILPATH)ccode $(MD).c $(PATH)merge.out

$(PATH)merge.out: $(MD)
        /lib/cpp $(OPTIONS) $(MD) | $(UTILPATH)metatmdl > $(MD).MD
        $(UTILPATH)tmdl $(MD).MD = $(PATH)tmdl.out
        $(UTILPATH)analysis -m 20 $(PATH)tmdl.out \
                $(PATH)analysis.out1 $(PATH)analysis.out2 $(PATH)analysis.out3
        $(UTILPATH)merge $(PATH)tmdl.out $(PATH)merge.out \
                $(PATH)analysis.out1 $(PATH)analysis.out2 $(PATH)analysis.out3
```