The table driven code generator from the Amsterdam Compiler Kit

Hans van Staveren

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

It is possible to automate the process of compiler building to a great extent using collections of tools. The Amsterdam Compiler Kit is such a collection of tools. This document provides a description of the internal workings of the table driven code generator in the Amsterdam Compiler Kit, and a description of syntax and semantics of the driving table.

>>> NOTE <<<
This document pertains to the **old** code generator. Refer to the "Second Revised Edition" for the new code generator.

Nov 1984

The table driven code generator from the Amsterdam Compiler Kit

Hans van Staveren

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

1. Introduction

Part of the Amsterdam Compiler Kit is a code generator system consisting of a code generator generator (*cgg* for short) and some machine independent C code. *Cgg* reads a machine description table and creates two files, tables.h and tables.c. These are then used together with other C code to produce a code generator for the machine at hand.

This in turn reads compact EM code and produces assembly code. The remainder of this document will first broadly describe the working of the code generator, then a description of the machine table follows after which the internal workings of the code generator will be explained.

The reader is assumed to have at least a vague notion about the semantics of the intermediary EM code. Someone wishing to write a table for a new machine should be thoroughly acquainted with EM code and the assembly code of the machine at hand.

2. Global overview of the workings of the code generator.

The code generator or *cg* tries to generate good code by simulating the runtime stack of the program compiled and delaying emission of code as long as possible. It also keeps track of register contents, which enables it to eliminate redundant moves, and tries to eliminate redundant tests by keeping information about condition code status, if applicable for the machine.

Cg maintains a 'fakestack' containing 'tokens' that are built by executing the pseudo code contained in the code rules given by the table writer. One can think of the fakestack as a logical extension of the real stack the program compiled will have when run. During code generation tokens will be kept on the fakestack as long as possible but when they are moved to the real stack, by generating code for the push, all tokens above* the tokens pushed will be pushed also, so that the fakestack will not contain holes.

The main loop of *cg* is this:

1) find a pattern of EM instructions starting at the current one to generate code for. This pattern will usually be of length one but longer patterns can be used.
2) Select one of the possibly many stack patterns that go with this EM pattern on the basis of heuristics and/or lookahead.
3) Force the current fakestack contents to match the pattern. This may involve copying tokens to registers, making dummy transformations, e.g. to transform a "local" into an "register offsetted" or might even cause to have the complete fakestack contents put to the real stack and then back into registers if no suitable transformations were provided by the table writer.
4) Execute the pseudocode associated with the code rule just selected, this may cause registers to be allocated, code to be emitted etc..
5) Put tokens onto the fakestack to reflect the result of the operation.
6) Insert some EM instructions into the stream, this is possible but not common.

---

* in the rest of this document the stack is assumed to grow downwards, although the top of the stack will mean the first element that will be popped.

7) Account for the cost. The cost is kept in a (space, time) vector and lookahead decisions are based on a linear combination of these.

The table that drives *cg* is not read in every time, but instead is used at compiletime of *cg* to set parameters and to load pseudocode tables. A program called *cgg* reads the table and produces large lists of numbers that are compiled together with machine independent code to produce a code generator for the machine at hand.

3. Description of the machine table

The machine description table consists of the following sections:

1) Constant definitions
2) Register definitions
3) Token definitions
4) Token expression definitions
5) Code rules
6) Move definitions
7) Test definitions
8) Stacking definitions

Input is in free format, white space and newlines may be used at will to improve legibility. Identifiers used in the table have the same syntax as C identifiers, upper and lower case considered different, all characters significant. There is however one exception: identifiers must be more than one character long for parsing reasons. C style comments are accepted

```
/* this is a comment */
```

and #define macros may be used if the need arises.

3.1. Some constants

Before anything else three constants must be defined, all with the syntax NAME=value, value being an integer. These constants are:

EM_WSIZE
: Number of bytes in a machine word. This is the number of bytes a simple **loc** instruction will put on the stack.

EM_PSIZE
: Number of bytes in a pointer. This is the number of bytes a **lal** instruction will put on the stack.

EM_BSIZE
: Number of bytes in the hole between AB and LB. If the calling sequence just saves PC and LB this size will be twice the pointersize.

EM_WSIZE and EM_PSIZE are checked when a program is compiled with the resulting code generator. EM_BSIZE is used by *cg* to add to the offset of instructions dealing with locals having positive offsets, i.e. parameters.

Optionally one can give here the factors with which the size and time parts of the cost function have to be multiplied to ensure they have the same order of magnitude. This can be done as

$$\text{TIMEFACTOR} = C_1/C_2$$
$$\text{SIZEFACTOR} = C_3/C_4$$

Above numbers must be read as rational numbers. Defaults are 1/1 for both of them. These constants set the default size/time tradeoff in the code generator, so if TIMEFACTOR and SIZEFACTOR are both 1 the code generator will choose at random between two codesequences where one has cost (10,4) and the other has cost (8,6). See also the description of the cost field below.

Also optional is the definition of a printformat for integers in the codefile. This is given as

FORMAT = string

The default for string is "%ld". For example on the PDP 11 one can use

FORMAT= "0%lo"

to satisfy the old UNIX assembler that reads octal unless followed by a period, and the ACK assembler that follows C conventions.

3.2. Register definition

The next part of the tables describes the various registers of the machine and defines identifiers to be used in later parts of the tables. Example for the PDP-11:

```
REGISTERS:
R0 = ( "r0",2), REG.
R1 = ( "r1",2), REG, ODDREG.
R2 = ( "r2",2), REG.
R3 = ( "r3",2), REG, ODDREG.
R4 = ( "r4",2), REG.
LB = ( "r5",2), LOCALBASE.
R01= ( "r0",4,R0,R1), REGPAIR.
R23= ( "r2",4,R2,R3), REGPAIR.
FR0= ( "r0",4), FREG.
FR1= ( "r1",4), FREG.
FR2= ( "r2",4), FREG.
FR3= ( "r3",4), FREG.
DR0= ( "r0",8,FR0), DREG.
DR1= ( "r1",8,FR1), DREG.
DR2= ( "r2",8,FR2), DREG.
DR3= ( "r3",8,FR3), DREG.
```

The identifier before the '=' sign is the name of the register as used further on in the table. The string is the name of the register as far as the assembler is concerned. The number is the size of the register in bytes. Identifiers following the number but within the parentheses are previously defined registernames that are contained in the register being defined. The identifiers following the closing parenthesis are properties of the register. So for example R23 is a register with assembler name r2, 4 bytes long, contains the registers R2 and R3 and has the property REGPAIR.

It might seem wise to list each and every property of a register, so one might give R0 the extra property MFPTREG named after the not too well known MFPT instruction on newer PDP-11 types, but this is not a good idea. Every extra property means the registerset is more unorthogonal and *cg* execution time is influenced by that, because it has to take into account a larger set of registers that are not equivalent.

There is a predefined property SCRATCH that is dynamic, i.e. a register can have the property SCRATCH one time, and loose it the next. A register has the property SCRATCH when it has a reference count of one. One needs to be able to discriminate between SCRATCH registers and others, because it is only allowed to do arithmetic on SCRATCH registers.

3.3. Stack token definition

The next part describes all possible tokens that can reside on the fakestack during code generation. Attributes of a token are described in the form of a C struct declaration, this is followed by the size in bytes of the token, optionally followed by the cost of the token when used as an addressing mode and the format to be used on output.

Tokens should usually be declared for every addressing mode of the machine at hand and for every size directly usable in a machine instruction. Example for the PDP-11 (incomplete):

```
TOKENS:
IREG2 =          { REGISTER reg; } 2 "*%[reg]" /* indirect register */
REGCONST =       { REGISTER reg; STRING off; } 2 /* not really addressable */
REGOFF2 =        { REGISTER reg; STRING off; } 2 "%[off](%[reg])"
IREGOFF2 =       { REGISTER reg; STRING off; } 2 "*%[off](%[reg])"
CONST =          { INT off; } 2 cost=(2,850) "$%[off]."
EXTERN2 =        { STRING off; } 2 "%[off]"
IEXTERN2 =       { STRING off; } 2 "*%[off]"
PAIRSIGNED =     { REGISTER regeven,regodd; } 2 "%[regeven]"
```

Types allowed in the struct are REGISTER, INT and STRING. Tokens without a printformat should never be output.

Notice that tokens need not correspond to addressing modes, the REGCONST token listed above, meaning the sum of the contents of the register and the constant, has no corresponding addressing mode on the PDP-11, but is included so that a sequence of add constant, load indirect, can be handled efficiently. This REGCONST token is needed as part of the path

REGISTER -> REGCONST -> REGOFF

of which the first and the last "exist" and the middle is needed only as an intermediate step.

3.4. Token expressions

Usually machines have certain collections of addressing modes that can be used with certain instructions. The stack patterns in the table are lists of these collections and since it is cumbersome to write out these long lists every time, there is a section here to give names to these collections. Please note that it is not forbidden to write out a token expression in the remainder of the table, but for clarity it is usually better not to. Example for the PDP-11 (incomplete):

```
TOKENEXPRESSIONS:
SOURCE2 = REG + IREG2 + REGOFF2 + IREGOFF2 + CONST + EXTERN2 +
        IEXTERN2
SREG    = REG * SCRATCH
```

Permissible in the expressions are all PASCAL set operators, i.e.

+ set union

- set difference

* set intersection

Every tokenidentifier is also a token expression identifier denoting the singleton collection of tokens containing just itself. Every register property as defined above is also a token expression matching all registers with that property when on the fakestack. The standard token expression identifier ALL denotes the collection of all tokens.

3.5. Expressions

Throughout the rest of the table expressions can be used in some places. This section will give the syntax and semantics of expressions. There are four types of expressions: integer, string, register and undefined. Type checking is performed by *cgg*. An operator with at least one undefined operand returns undefined except for the defined() function mentioned below. An undefined expression is interpreted as FALSE when it is needed as a truth value. Basic terms in an expression are

number	A number is a constant of type integer.
string	A string within double quotes is a constant of type string. All the normal C style escapes may be used within the string.
REGIDENT	The name of a register is a constant of type register.

$*i*	A dollarsign followed by a number is the representation of the argument of EM instruction . The type of the operand is dependent on the instruction, sometimes it is integer, sometimes it is string. It is undefined when the instruction has no operand. Although an exhaustive list could be given describing all the types the following rule of thumb will suffice. If it is unimaginable for the operand of the instruction ever to be something different from a plain integer, the type is integer, otherwise it is string. *Cg* makes all necessary conversions, like adding EM_BSIZE to positive arguments of instructions dealing with locals, prepending underlines to global names, converting codelabels into a unique representation etc. Details about this can be found in the section about machine dependent C code.
%[1]	This in general means the token mentioned first in the stack pattern. When used inside an expression the token must be a simple register. Type of this is register.
%[1.off]	This means field "off" of the first stack pattern token. Type is the same as that of field "off". To use this expression implies a check that all tokens in the token expression used have the same attributes.
%[1.1]	This is the first subregister of the first token. Previous comments apply.
%[b]	The second allocated register.
%[a.2]	The second subregister of the first allocated register.

All normal C operators apply to integers, the + operator serves for string concatenation and register expressions can only be compared to each other. Furthermore there are some special "functions":

tostring(e)	Converts an integer expression e to a string.
defined(e)	Returns 1 if expression e is defined, 0 otherwise.
samesign(e1,e2)	Returns 1 if integer expression e1 and e2 have the same sign.
sfit(e1,e2)	Returns 1 if integer expression e1 fits as a signed integer into a field of e2 bits, 0 otherwise.
ufit(e1,e2)	Same as above but now for unsigned e1.
rom(a,n)	Integer expression giving the n'th argument from the **rom** descriptor pointed at by the a'th EM instruction. Undefined if that descriptor does not exist.
loww(a)	Returns the lower half of the argument of the a'th EM instruction. This is used to split the arguments of a **ldc** instruction.
highw(a)	Same for upper half.

3.6. Code rules

The largest section of the tables consists of the code generation rules. They specify EM patterns, stack patterns, code to be generated etc. Syntax is

code rule : EM pattern 'I' stack pattern 'I' code 'I'
stack replacement 'I' EM replacement 'I' cost ;

All parts are optional, however there must be at least one pattern present. If the empattern is missing the rule becomes a rewriting rule or *coercion* to be used when code generation cannot continue because of an invalid stack pattern. The code rules are preceded by the word

CODE:

The next paragraphs describe the various parts in detail.

3.6.1. The EM pattern

The EM pattern consists of a list of EM mnemonics followed by a boolean expression. Examples:

loe

will match a single **loe** instruction,

loc loc cif $1==2 && $2==8

is a pattern that will match

loc 2
loc 8
cif

and

lol inc stl $1==$3

will match for example

lol 6	**lol** -2		**lol** 4
inc	**inc**	but *not*	**inc**
stl 6	**stl** -2		**stl** -4

A missing boolean expression evaluates to TRUE.

When the EM pattern is the same as in the previous code rule the pattern should be given as '...'. The code generator will match the longest EM pattern on every occasion, if two patterns of the same length match the first in the table will be chosen, while all patterns of length greater than or equal to three are considered to be of the same length.

3.6.2. The stack pattern

The stack pattern is a list of token expressions, usually token expression identifiers for clarity. No boolean expression is allowed here. The first expression is the one that matches the top of the stack.

The pattern can be followed by the word STACK in which case the pattern only matches if there is nothing else on the fakestack. The code generator will stack everything not matched at the start of the rule.

The pattern can be preceded with the word

nocoercions:

which tells the code generator not to try to coerce to the pattern but only to use it when it is already there. There are two reasons for this construction, correctness and speed. It is needed for correctness when the pattern contains a register that is not transparent when data is moved through it.

Example: on the PDP-11 the shortest code for

lae a
loi 8
lae b
sti 8

is

movf _a,fr0
movf fr0,_b

assuming that the floating point processor is in double precision mode and fr0 is free. Unfortunately this is not correct since a trap can occur on certain kinds of data. This could happen if there was a pattern for **sti** 8 that allowed one to move a floating point register not preceded by nocoercions: . The code generator would then find that moving the 8-byte global _a to a floating point register and then storing it to _b was the cheapest, assuming that the space/time knob was turned far enough to space. It is unfortunate that the type information is no longer present, since if _a really is a floating point number the move could be made without error.

The second reason for the nocoercions: construct is speed. When the code generator has a long list of possible stack patterns for one EM pattern it can waste a lot of time trying to find coercions to all of them, while the mere presence of such a long list indicates that the table writer has given a lot of special cases. In this case prepending all the special cases by nocoercions: will stop the code generator from trying to find

things there aren't.

3.6.3. The code part

The code part consists of three parts, stack cleanup, register allocation and code to generate. All of these may be omitted.

3.6.3.1. Stack cleanup

The stack cleanup part describes certain stacktokens that should neither remain on the fakestack, nor remembered as contents of registers. This is usually only required with store operations. The entire fakestack, except for the part matched in the stack pattern, is searched for tokens matching the expression and they are copied to the real stack. Every register that contains the stacktoken is marked as empty.

Syntax is

remove(token expression) *or*
remove(token expression, boolean expression)

Example:

remove(REGOFF2,%[reg] != LB || %[off] == $1)

is part of a remove() call for use in the **stl** code rule. It removes all register offsetted tokens where the register is not the localbase plus the local wherein the store is done. The necessity for this can be seen from the following example:

lol 4
inl 4
stl 6

Without a proper remove() call in the rule for **inl** code would be generated as here

inc 4(r5)
mov 4(r5),6(r5)

so local 6 would be given the new value of local 4 instead of the old as the EM code prescribed.

When generating something like a branch instruction it might be needed to empty the fakestack completely. This can of course be done with

remove(ALL)

3.6.3.2. Register allocation

The register allocation part describes the kind of registers needed. Syntax for allocate() is

allocate(itemlist)

where itemlist is a list of three kinds of things:

1) a tokendescription, for example %[1].
This will instruct the code generator to temporarily decrement the reference count of all registers contained in the token, so that they are available for allocation in this allocate() call if they were only used in that token. See example below.

2) a register property.
This will allocate a register with that property. The register will be marked as empty at this point. Lookahead will be performed if necessary.

3) a register property with initialization.
This will allocate the register as in 2) but will also initialize it. This eases the task of the code generator because it can find a register already filled with the right value if it exists.

Examples:

allocate(OREG)

will allocate an odd register, while

allocate(REG={REGOFF2,LB,$1})

will allocate a register while simultaneously filling it with the asked value.
Inside the coercion from SOURCE2 to REGISTER in the PDP-11 table the following allocate() can be found.

allocate(%[1],REG=%[1])

This tells the code generator that registers contained in %[1] can be used again and asks to fill the register allocated with %[1]. So if %[1]={REGOFF2,R3,"4"} and R3 has a reference count of 1 the following code might be generated.

mov 4(r3),r3

In the rest of the line the registers allocated can be named by %[a] and %[b.1],%[b.2], i.e. with lower case letters in order of allocation.

Warning:

allocate(R3)

is not the way to allocate R3. R3 is not a register property, so it will be seen as a token description and the effect is that R3 will have its reference count decremented.

3.6.3.3. Code

Code to be generated is specified as a list of items of the following kind:

1) a string in double quotes ("This is a string").
This is copied to the codefile and a newline (\n) is appended. Inside the string all normal C string conventions are allowed, and substitutions can be made of the following sorts.
   a) $1, $2 etc. These are the operands of the corresponding EM instructions and are printed according to their type. To put a real '$' inside the string it must be doubled ('$$').
   b) %[1], %[2.reg], %[b.1] etc. These have their obvious meaning. If they describe a complete token (%[1]) the printformat for the token is used. If they stand for a basic term in an expression they will be printed according to their type. To put a real '%' inside the string it must be doubled ('%%').
   c) %(arbitrary expression %). This allows inclusion of arbitrary expressions inside strings. Usually not needed very often, so that the awkward notation is not too bad. Note that %(%[1]%) is equivalent to %[1].
2) a move() call. This has the following syntax:

   move(token description, token description)

   Moves are handled specially since that enables the code generator to keep track of register contents. Example:

   move(R3,{REGOFF2,LB,$1})

   will generate code to move R3 to $1(r5) except when R3 already was a copy of $1(r5). Then the code will be omitted. The rules describing how to move things to each other can be found in the MOVES section described below.
3) an erase() call. This has the following syntax:

   erase(register expression)

   This tells the code generator that the register mentioned no longer has any useful value. This is *necessary* after code in the table has changed the contents of registers. For example, after an add to a register the register must be erased, because the contents do no longer match any token.

4) For machines that have condition codes, alas most of them do, there are provisions to remember condition code setting and prevent needless testing. To set the condition code to a token put in the code the following call:

test(token)

where token can be all of the standard forms that can also be used in move(). This will generate a test if the condition codes were not already set to that token. It is also possible to tell *cg* that a certain operation, like a preceding add has set the condition codes to some token with the call

setcc(token)

So a sequence of a setcc and a test on the same token will generate no code. Another allowed call within the code is

samecc

which tells the code generator that condition codes were unaffected in this rule. If no setcc or samecc has been given the default is

nocc

when a piece of code contained strings, which tells the code generator that the condition codes have no useful value any more.

3.6.4. Stack replacement

The stack replacement is a possibly empty list of items to be pushed onto the fakestack. Three kinds of items are possible:

1) An item of the form %[1]. This will push the stacktoken mentioned back onto the stack unchanged.
2) A register expression. This will push the register mentioned onto the fakestack.
3) An item of the form { REGOFF2,%[1.reg],$1 }. This generates a token with tokenidentifier REGOFF2 and attributes in order of declaration.

All tokens matched by the stack pattern at the beginning of the code rule are first removed and their registers deallocated. Items are pushed in the order of appearance. This means that the last item will be on the top of the stack after the push. So if the stack pattern contained two token expressions and they must be pushed back unchanged, they have to be specified as stack replacement

%[2] %[1]

and not the other way around.

3.6.5. EM replacement

In exceptional cases it might be useful to leave part of an empattern undone. For example, a **sdl** instruction might be split into two **stl** instructions when there is no 4-byte quantity on the stack. The emreplacement part allows one to express this. Example:

stl $1 **stl** $1+2

The instructions are inserted in the stream so that they can match the first part of a pattern in the next step. Note that since the code generator traverses the EM instructions in a strict linear fashion, it is impossible to let the EM replacement match later parts of a pattern. So if there is a pattern

loc stl $1==0

and the input is

loc 0 **sdl** 4

the **loc** 0 will be processed first, then the **sdl** might be split into two **stl**'s but the pattern cannot match now.

3.6.6. Cost

The cost field can be specified when there is more than one code rule with the same empattern. If the code generator has a choice between two possibilities to generate code it will choose the cheapest according to the cost field. The cost for a code generation is the sum of the costs of all the coercions needed, plus the cost for freeing registers plus the cost of the code rule itself.

The format of the costfield is

(nbytes, time) or
(nbytes, time) + %[*i*]

with time in the metric desired, like nanoseconds or states. See constants section above. The %[*i*] in the second example is used for adding the cost of a certain address mode used in the code generated. This can of course be repeated if desired. The cost of the address mode must then be specified in the token definition section.

3.6.7. Examples

A list of examples for the PDP-11 is given here. Far from being complete it gives examples of most kinds of instructions.

```
adi $1==2 | SREG,SOURCE2 |
        "add %[2],%[1]" erase(%[1]) setcc(%[1])
         | %[1] | | (2,450) + %[2]
...     | SOURCE2,SREG |
        "add %[1],%[2]" erase(%[2]) setcc(%[2])
         | %[2] | | (2,450) + %[1]
```

is an example of the use of the '...' construct and shows how to place erase() and setcc() calls.

```
dvi $1==2 | SOURCE2,SPAIRSIGNED |
        "div %[1],%[2]" erase(%[2])
         | %[2.regeven] | |

cmi tgt $1==2 | SOURCE2,SOURCE2 | allocate(REG={CONST,0})
        "cmp %[2],%[1];ble 1f;inc %[a];1:" erase(%[a])
         | %[a] | |

cal | STACK |
        "jsr pc,$1"
         | | |

lol | | | { REGOFF2, LB, $1 } | |

stl | SOURCE2 |
        remove(REGOFF2,%[off]==$1)
        move(%[1],{REGOFF2,LB,$1})
         | | |

| SOURCE2 |
        allocate(%[1],REGPAIR)
        move(%[1],%[a.2])
        test(%[a.2])
        "sxt %[a.even]" | { PAIRSIGNED, %[a.1], %[a.2] }| |
```

This coercion shows how to use the move and test calls. At first one might think that the testcall is unnecessary, since the move will have set the condition codes, but the move may never have been executed if the register already contained the value, in which case it is necessary to do the test. If the move was executed

the test will be omitted.

| SOURCE2 | allocate(%[1],REG=%[1]) | %[a] | |

sdl | SOURCE2 | | %[1] | **stl** $1 **stl** $1+2 |

exg $1==2 | SOURCE2 SOURCE2 | | %[1] %[2] | |

This last example again shows the difference in the order of the stack pattern and the stack replacement.

3.7. Move code rules

When issuing a move() call as described above or a register allocation with initialization, the code generator has to know which instruction to use for the move. The code will of course only be generated if it cannot be omitted. This is listed in the move section of the tables by giving a list of tuples:

(source, destination, codepart [, costfield])

where the square brackets mean the costfield is optional. Example for the PDP-11

```
MOVES:
( CONST %[off]==0 , SOURCE2, "clr %[2]" )
( SOURCE2, SOURCE2, "mov %[1],%[2]" )
```

The moves are scanned from top to bottom, so the first one that matches will be chosen.

3.8. Test code rules

When issuing a test() call as described above, the code generator has to know which instruction to use for the test. The code will only be generated if the condition codes were not already set to the token. This is listed in the test section of the tables by giving a list of tuples:

(source, codepart [, costfield])

Example for the PDP-11

```
TESTS:
( SOURCE2, "tst %[1]")
( DREG, "tstf %[1]\ncfcc")
```

The tests are scanned from top to bottom, so the first one that matches will be chosen.

3.9. Stacking code rules.

When the code generator has to stack a token it must know which code to use. Since it must at all times be possible to empty the fakestack even when no registers are free, it is mandatory that all tokens used must have a rule attached for stacking them without using a scratch register. Since however this might be clumsy and a register might in practice be available it is also possible to give rules which use a register. On the Intel 8086 for example, there is no instruction to push a constant without using a register, and the code needed to do it without, must use global data and as such is very complicated and wasteful of memory and time. It can therefore be left to be used in extreme cases, while in general the constant is pushed through a register. The stacking rules are listed in the stack section of the table as a list of tuples:

(source, [register property] , codepart [, costfield])

Example for the Intel 8086:

```
STACKS:
(CONST, REG, move(%[1],%[a]) "push %[a]")
(REG ,, "push %[1]")
```

4. The files mach.h and mach.c

The table writer must also supply two files containing machine dependent declarations and C code. These files are mach.h and mach.c.

4.1. Types in the code generator

Three different types of integer coexist in the code generator and their range depends on the machine at hand. The type 'int' is used for things like labelcounters that won't require more than 16 bits precision. The type 'word' is used among others to assemble datawords and is of type 'long'. The type 'full' is used for addresses and is of type 'long' if EM_WSIZE>2 or EM_PSIZE>2.

In macro and function definitions in later paragraphs implicit typing will be used for parameters, that is parameters starting with an 's' will be of type string, and the letters 'i','w','f' will stand for int, word and full respectively.

4.2. Global variables to work with

Some global variables are present in the code generator that can be manipulated by the routines in mach.h and mach.c.

The declarations are:

```
FILE *codefile;          /* code is emitted on this stream */
word part_word;          /* words to be output are put together here */
int part_size;           /* number of bytes already put in part_word */
char str[];              /* Last string read in */
long argval;             /* Last int read and kept */
```

4.3. Macros in mach.h

In the file mach.h a collection of macros is defined that have to do with formatting of assembly code for the machine at hand. Some of these macros can of course be left undefined in which case the macro calls are left in the source and will be treated as function calls. These functions can then be defined in *mach.c*.

The macros to be defined are:

ex_ap(s)	Must print the magic incantations that will mark the symbol to be exported to other modules. This is the translation of the EM **exa** and **exp** instructions.
in_ap(s)	Same to import the symbol. Translation of **ina** and **inp**.
newplb(s)	Must print the definition of procedure label *s*. If left undefined the newilb() macro is used instead.
newilb(s)	Must print the definition of instruction label *s*.
newdlb(s)	Must print the definition of data label *s*.
dlbdlb(s1,s2)	Must define data label *s1* to be equal to *s2*.
newlbss(s,f)	Must declare a piece of memory initialized to BSS_INIT(see below) of length *f* and with label *s*.
cst_fmt	Format to be used when converting constant arguments of EM instructions to string. Argument to be formatted will be 'full'.
off_fmt	Format to be used for integer part of label+constant, argument will be 'full'.
fmt_ilb(ip,il,s)	Must use the numbers *ip* and *il* which are a procedure number and a label number respectively and copy a string to *s* that must be unique for that combination. This procedure is optional, if it is not given ilb_fmt must be defined as below.
ilb_fmt	Format to be used for creation of unique instruction labels. Arguments will be a unique procedure number (int) and the label number (int).

dlb_fmt	Format to be used for printing numeric data labels. Argument will be 'int'.
hol_fmt	Format to be used for generation of labels for space generated by a **hol** pseudo. Argument will be 'int'.
hol_off	Format to be used for printing of the address of an element in **hol** space. Arguments will be the offset in the **hol** block (word) and the number of the **hol** (int).
con_cst(w)	Must generate output that will assemble into one machineword.
con_ilb(s)	Must generate output that will put the address of the instruction label into the datastream.
con_dlb(s)	Must generate output that will put the address of the data label into the datastream.
fmt_id(sf,st)	Must take the string in *sf* which is a nonnumeric global label, and transform it into a copy made to *st* which will not collide with reserved assembler words and system labels. This procedure is optional, if it is not given the id_first macro is used as defined below.
id_first	Must be a character. This is prepended to all nonnumeric global labels if their length is shorter than the maximum allowed(currently 8) or if they already start with that character. This is to avoid conflicts of user labels with system labels.
BSS_INIT	Must be a constant. This is the value filled in all the words not initialized explicitly. This is loader and system dependent. If omitted no initialization is assumed.

4.3.1. Example mach.h for the PDP-11

```
#define ex_ap(y)    fprintf(codefile,"\t.globl %s\n",y)
#define in_ap(y)    /* nothing */

#define newplb(x)   fprintf(codefile,"%s:\n",x)
#define newilb(x)   fprintf(codefile,"%s:\n",x)
#define newdlb(x)   fprintf(codefile,"%s:\n",x)
#define             dlbdlb(x,y)         fprintf(codefile,"%s=%s\n",x,y)
#define newlbss(l,x)                    fprintf(codefile,"%s:.=.+%d.\n",l,x);

#define cst_fmt                         "$%d."
#define off_fmt                         "%d."
#define ilb_fmt                         "I%x_%x"
#define dlb_fmt                         "_%d"
#define             hol_fmt                                 "hol%d"

#define hol_off                         "%ld.+hol%d"

#define con_cst(x)  fprintf(codefile,"%ld.\n",x)
#define con_ilb(x)  fprintf(codefile,"%s\n",x)
#define con_dlb(x)  fprintf(codefile,"%s\n",x)

#define id_first    '_'
#define BSS_INIT    0
```

4.4. Functions in mach.c

In mach.c some functions must be supplied, mostly manipulating data resulting from pseudoinstructions. The specifications are given here, implicit typing of parameters as above.

con_part(isz,word)	This function must manipulate the globals part_word and part_size to append the isz bytes contained in word to the output stream. If part_word is full, i.e. part_size==EM_WSIZE the function part_flush() may be called to empty the

	buffer. This is the function that must go through the trouble of doing byte order in words correct.
con_mult(w_size)	This function must take the string str[] and create an integer from the string of size w_size and generate code to assemble global data for that integer. Only the sizes for which arithmetic is implemented need be handled, so if 200-byte integer division is not implemented, 200-byte integer global data do not have to be implemented. Here one must take care of word order in long integers.
con_float()	This function must generate code to assemble a floating point number of which the size is contained in argval and the ASCII representation in str[].
prolog(f_nlocals)	This function is called at the start of every procedure. Function prolog code must be generated, and room made for local variables for a total of f_nlocals bytes.
mes(w_mesno)	This function is called when a **mes** pseudo is seen that is not handled by the machine independent part. The example below probably shows all the table writer ever has to know about that.
segname[]	This is not a function, but an array of four strings. These strings are put out whenever the code generator switches segments. Segments are SEGTXT, SEGCON, SEGROM and SEGBSS in that order.

4.4.1. Example mach.c for the PDP-11

As an example of the sort of code expected, the mach.c for the PDP-11 is presented here.

```
/*
 * machine dependent back end routines for the PDP-11
 */

con_part(sz,w) register sz; word w; {

                        while (part_size % sz)
                                        part_size++;
                        if (part_size == EM_WSIZE)
                                        part_flush();
                        if (sz == 1) {
                                        w &= 0xFF;
                                        if (part_size)
                                                        w <<= 8;
                                        part_word |= w;
                        } else {
                                        assert(sz == 2);
                                        part_word = w;
                        }
                        part_size += sz;
}

con_mult(sz) word sz; {
                        long l;

                        if (sz != 4)
                                        fatal("bad icon/ucon size");
                        l = atol(str);
                        fprintf(codefile,"\t%o;%o\n",(int)(l>>16),(int)l);
}

con_float() {
                        double f;
                        register short *p,i;

                        /*
                         * This code is correct only when the code generator is
                         * run on a PDP-11 or VAX-11 since it assumes native
                         * floating point format is PDP-11 format.
                         */

                        if (argval != 4 && argval != 8)
                                        fatal("bad fcon size");
                        f = atof(str);
                        p = (short *) &f;
                        i = *p++;
                        if (argval == 8) {
                                        fprintf(codefile,"\t%o;%o;",i,*p++);
                                        i = *p++;
                        }
                        fprintf(codefile,"\t%o;%o\n",i,*p++);
}

prolog(nlocals) full nlocals; {
```

```
                fprintf(codefile,"mov r5,-(sp)\nmov sp,r5\n");
                if (nlocals == 0)
                        return;
                if (nlocals == 2)
                        fprintf(codefile,"tst -(sp)\n");
                else
                        fprintf(codefile,"sub $%d.,sp\n",nlocals);
}

mes(type) word type; {
                int argt ;

                switch ( (int)type ) {
                case ms_ext :
                        for (;;) {
                                switch ( argt=getarg(
                                   ptyp(sp_cend)|ptyp(sp_pnam)|sym_ptyp) ) {
                                case sp_cend :
                                        return ;
                                default:
                                        strarg(argt) ;
                                        fprintf(codefile,".globl %s\n",argstr) ;
                                        break ;
                                }
                        }
                default :
                        while ( getarg(any_ptyp) != sp_cend ) ;
                        break ;
                }
}

char    *segname[] = {
                ".text",        /* SEGTXT */
                ".data",        /* SEGCON */
                ".data",        /* SEGROM */
                ".bss"          /* SEGBSS */
};
```

5. Coercions

A central part in code generation is taken by the *coercions*. It is the responsibility of the table writer to provide all necessary coercions so that code generation can continue. The very minimal set of coercions are the coercions to unstack every token expression, in combination with the rules to stack every token.

If these are present the code generator can always make the necessary transformations by stacking and unstacking. Of course for codequality it is usually best to provide extra coercions to prevent this stacking to take place. *Cg* discriminates three types of coercions:

1) Unstacking coercions. This category can use the allocate() call in its code.

2) Splitting coercions, these are the coercions that split larger tokens into smaller ones.

3) Transforming coercions, these are the coercions that transform a token into another one of the same size. This category can use the allocate() call in its code.

When a stack configuration does not match the stack pattern *coercions* are searched for in the following order:

1) First tokens are split if necessary to get their sizes right.

2) Then transforming coercions are found that will make the pattern match.

3) Finally if the stack pattern is longer than the fakestack contents unstacking coercions will be used to fill up the pattern.

At any point, when coercions are missing so code generation could not continue, the offending tokens are stacked.

6. Internal workings of the code generator.

6.1. Description of tables.c and tables.h contents

In this section the intermediate files will be described that are produced by *cgg* and compiled with machine independent code to produce a code generator.

6.1.1. Tables.c

Tables.c contains a large number of initialized array's of all sorts. Description of each follows:

byte code rules[]
: Pseudo code interpreted by the code generator. Always starts with some opcode followed by operands depending on the opcode. Integers in this table are between 0 and 32767 and have a one byte encoding if between 0 and 127.

char stregclass[]
: Number of computed static register class per register. Two registers are in the same class if they have the same properties and don't share a common subregister.

struct reginfo machregs[]
: Info per register. Initialized with representation string, size, members of the register and set of registers affected when this one is changed. Also contains room for runtime information, like contents and reference count.

tkdef_t tokens[]
: Information per tokentype. Initialized with size, cost, type of operands and formatstring.

node_t enodes[]
: List of triples representing expressions for the code generator.

string code strings[]
: List of strings. All strings are put in a list and checked for duplication, so only one copy per string will reside here.

set_t machsets[]
: List of token expression sets. Bit 0 of the set is used for the SCRATCH property of registers, bit 1 upto NREG are for the corresponding registers and bit NREG+1 upto the end are for corresponding tokens.

inst_t tokeninstances[]
: List of descriptions for building tokens. Contains type of rule for building one, plus operands depending on the type.

move_t moves[]
: List of move rules. Contains token expressions for source and destination plus cost and index for code rule.

byte pattern[]
: EM patterns. This is structured internally as chains of patterns, each chain pointed at by pathash[]. After each pattern the list of possible code rules is given.

int pathash[256]
: Indices into pattern[] for all patterns with a certain low order byte of the hashing function.

c1_t c1coercs[]
: List of rules to stack tokens. Contains token expressions, register needed, cost and code rule.

c2_t c2coercs[]
: List of splitting coercions. Token expressions, split factor, replacements and code rule.

c3_t c3coercs[]
: List of one to one coercions. Token expressions, register needed, replacement and code rule.

struct reginfo **reglist[]
: List of lists of pointers to register information. For every property the list is here to find the registers corresponding to it.

6.1.2. tables.h

In tables.h various derived constants for the tables are given. They are then used to determine array sizes in the actual code generator, plus loop termination in some cases.

6.2. Other important data structures

During code generation some other data structures are used and here is a short description of some of the important ones.

Tokens are kept in the code generator as a struct consisting of one integer *t_token* which is -1 if the token is a register, and the number of the token otherwise, plus an array of *TOKENSIZE* unions *t_att* of which the first is the register number in case of a register.

The fakestack is an array of these tokens, there is a global variable *stackheight*.

The results of expressions are kept in a struct *result* with elements *e_typ*, giving the type of the expression: *EV_INT*, *EV_REG* or *EV_STR*, and a union *e_v* which contains the real result.

6.3. A tour through the sources

6.3.1. codegen.c

The file codegen.c contains one large function consisting of one giant switch statement. It is the interpreter for the code generator pseudo code as contained in code rules[]. This function can call itself recursively when doing lookahead. Arguments are:

codep
: Pointer into code rules, pseudo program counter.

ply
: Number of EM pattern lookahead allowed.

toplevel
: Boolean telling whether this is the toplevel codegen() or a deeper incarnation.

costlimit
: A cutoff value to limit searches. If the cost crosses costlimit the incarnation can terminate.

forced
: A register number if nonzero. This is used inside coercions to force the allocate() call to allocate a register determined by earlier lookahead.

The instructions inplemented in the switch:

6.3.1.1. DO_NEXTEM

Matches the next EM pattern and does lookahead if necessary to find the best code rule associated with this pattern. Heuristics are used to determine best code rule when possible. This is done by calling the distance() function.

6.3.1.2. DO_COERC

This sets the code generator in the state to do a from stack coercion.

6.3.1.3. DO_XMATCH

This is done when a match no longer has to be checked. Used when the nocoercions: trick is used in the table.

6.3.1.4. DO_MATCH

This is the big one inside this function. It has the task to transform the contents of the current fakestack to match the pattern given after it.

Since the code generator does not know combining coercions, i.e. there is no way to make a big token out of two smaller ones, the first thing done is to stack every token that is too small. After that all tokens too big are split if possible to the right size.

Next the coercions are sought that would transform tokens in place to the right one, plus the coercions that would pop tokens of the stack. Each of those might need a register, so a list of registers is generated and at the end of looking for coercions the function *tuples()* is called to generate the list of all possible *n*-tuples, where *n* equals the number of registers needed.

Lookahead is now performed if the number of tuples is greater than one. If no possibility is found within the costlimit, the fakestack is made smaller by pushing the bottom token, and this process is repeated until either a way is found or the fakestack is completely empty and there is still no way to make the match.

If there is a way the corresponding coercions are executed and the code is finished.

6.3.1.5. DO_REMOVE

Here the remove() call is executed, all tokens matched by the token expression plus boolean expression are pushed. In the current implementation there is no attempt to move those tokens to registers, but that is a possible future extension.

6.3.1.6. DO_DEALLOCATE

This one temporarily decrements by one the reference count of all registers contained in the token given as argument.

6.3.1.7. DO_REALLOCATE

Here all temporary deallocates are made undone.

6.3.1.8. DO_ALLOCATE

This is the part that allocates a register and decides which one to use. If the *forced* argument was given its task is simple, otherwise some work must be done. First the list of possible registers is scanned, all free registers noted and it is noted whether any of those registers is already containing the initialization. If no registers are available some fakestack token is stacked and the process is repeated.

After that if an exact match was found, the list of registers is reduced to one register matching exactly out of every register class. Now lookahead is performed if necessary and the register chosen. If an initialization was given the corresponding move is performed, otherwise the register is marked empty.

6.3.1.9. DO_LOUTPUT

This prints a string and an expression. Only done on toplevel.

6.3.1.10. DO_ROUTPUT

Prints a string and a new line. Only on toplevel.

6.3.1.11. DO_MOVE

Calls the move() function in the code generator to implement the move() function in the table.

6.3.1.12. DO_ERASE

Marks the register that is its argument as empty.

6.3.1.13. DO_TOKREPLACE

This is the token replacement part. It is also called if there is no token replacement because it has some other functions as well.

First the tokens that will be pushed on the fakestack are computed and stored in a temporary array. Then the tokens that were matched in this rule are popped and their embedded registers have their reference

count decremented. After that the replacement tokens are pushed.

Finally all registers allocated in this rule have their reference count decremented. If they were not pushed on the fakestack they will be available again in the next code rule.

6.3.1.14. DO_EMREPLACE

Places replacement EM instructions back into the instruction stream.

6.3.1.15. DO_COST

Accounts for cost as given in the code rule.

6.3.1.16. DO_RETURN

Returns from this level of codegen(). Is used at the end of coercions, move rules etc..

6.3.2. compute.c

This module computes the various expressions as given in the enodes[] array. Nothing very special happens here, it is just a recursive function computing leaves of expressions and applying the operator.

6.3.3. equiv.c

In this module the tuples() function is implemented. It is given the number of registers needed and a list of register lists and it constructs a list of tuples where the *n*'th register comes from the *n*'th list. Before the list is constructed however the dynamic register classes are computed. Two registers are in the same dynamic class if they are in the same static class and their contents is the same.

After that the permute() recursive function is called to generate the list of tuples. After construction a generated tuple is added to the list if it is not already pairwise in the same class or if the register relations are not the same, i.e. if the first and second register share a common subregister in one tuple and not in the other they are considered different.

6.3.4. fillem.c

This is the routine that does the reading of EM instructions and the handling of pseudos. The mach.c module provided by the table writer is included at the end of this module. The routine fillemlines() is called by nextem() at toplevel to make sure there are enough instruction to match. It fills the EM instruction buffer up to 5 places from the end to keep room for EM replacement instructions, or up to a pseudo.

The dopseudo() function performs the function of the pseudo last encountered. If the pseudo is a **rom** the corresponding label is saved with the contents of the **rom** to be available to the code generator later. The rest of the routines are small service routines for either input or data output.

6.3.5. gencode.c

This module contains routines called by codegen() to generate the real code to the codefile. The function gencode() gets a string as argument and copies it to codefile while processing certain embedded control characters implementing the $2 and [1.reg] escapes. The function genexpr() prints the expression given as argument. It is used to implement the %(expr %) escape. The prtoken() function interprets the tokenformat as given in the tokens[] array.

6.3.6. glosym.c

This module maintains a list of global symbols that have a **rom** pseudo associated. There are functions to enter a symbol and to find a symbol.

6.3.7. main.c

Main routine of the code generator. Processes arguments and flags. Flags available are:

-d Sets debug mode if the code generator was not compiled with the NDEBUG macro defined. Debug mode gives very long output on stderr indicating all steps of the code generation process including

nesting of the codegen() function.

-p*n* Sets the lookahead depth to *n*, the *p* stands for ply, a well known word in chess playing programs.

-w*n* Sets the weight percentage for size in the cost function to *n* percent. Uses Euclides algorithm to simplify rationals.

6.3.8. move.c

Function to implement the move() pseudo function in the tables, register initialization and the setcc and test pseudo functions. First tests are made to try to prevent the move from really happening. The condition code register is treated special here. After that, if there is an after that, the move rule is found and the code executed.

6.3.9. nextem.c

The entry point of this module is nextem(). It hashes the next three EM instructions, and uses the low order byte of the hash as an index into the array pathash[], to find a chain of patterns in the array pattern[], that are all tried for a match.

The function trypat() does most of the work checking patterns. When a pattern is found to match all instructions the operands of the instruction are placed into the dollar[] array. Then the boolean expression is tried. If it matches the function can return, leaving the operands still in the dollar[] array, so later in the code rule they can still be used.

6.3.10. reg.c

Collection of routines to handle registers. Reference count routines are here, chrefcount() and getrefcount(), plus routines to erase a single register or all of them, erasereg() and cleanregs().

If NDEBUG hasn't been defined, here is also the routine that checks if the reference count kept with the register information is in agreement with the number of times it occurs on the fakestack.

6.3.11. salloc.c

Module for string allocation and garbage collection. Contains entry points myalloc(), a routine calling malloc() and checking whether room is left, myfree(), just free(), popstr() a function called from state.c to free all strings made since the last saved status. Furthermore there is salloc() which has the size of the string as parameter and returns a pointer to the allocated space, while keeping a copy of the pointer for garbage allocation purposes.

The function garbage_collect is called from codegen() at toplevel every now and then, and checks all places where strings may reside to mark strings as being in use. Strings not in use are returned to the pool of free space.

6.3.12. state.c

Set of routines called to save current status, restore a previous saved state and to free the room occupied by a saved state. A list of structs is kept here to save the state. If this is not done, small allocates will take space from the holes big enough for state saves, and as a result every new state save will need a new struct. The code generator runs out of room very rapidly under these conditions.

6.3.13. subr.c

Random set of leftover routines.

6.3.13.1. match

Computes whether a certain token matches a certain token expression. Just computes a bitnumber according to the algorithm explained with machsets[], and tests the bit and the boolean expression if it is there.

6.3.13.2. instance,cinstance

These two functions compute a token from a description. They differ very slight, cinstance() is used to compute the result of a coercion in a certain context and therefore has more arguments, which it uses instead of the global information instance() works on.

6.3.13.3. eqtoken

eqtoken computes whether two tokens can be considered identical. Used to check register contents during moves mainly.

6.3.13.4. distance

This is the heuristic function that computes a distance from the current fakestack contents to the token pattern in the table. It likes exact matches most, then matches where at least the sizes are correct and if the sizes are not correct it likes too large sizes more than too small, since splitting a token is easier than combining one.

6.3.13.5. split

This function tries to find a splitting coercion and executes it immediately when found. The fakestack is shuffled thoroughly when this happens, so pieces below the token that must be split are saved first.

6.3.13.6. docoerc

This function executes a coercion that was found. The same shuffling is done, so the top of the stack is again saved.

6.3.13.7. stackupto

This function gets a pointer into the fakestack and must stack every token including the one pointed at up to the bottom of the fakestack. The first stacking rule possible is used, so rules using registers must come first.

6.3.13.8. findcoerc

Looks for a one to one coercion, if found it returns a pointer to it and leaves a list of possible registers to use in the global variable curreglist. This is used by codegen().

6.3.14. var.c

Global variables used by more than one module. External definitions are in extern.h.