BEG – a Back End Generator

User Manual

Helmut Emmelmann

GMD Forschungsstelle an der Universität Karlsruhe
Haid-und-Neu-Str. 7, D–7500 Karlsruhe 1, Germany
Tel.: 49-721-6622-0
Fax : 49-721-6622-968
e-mail: emmel@karlsruhe.gmd.dbp.de

November 9, 1989, Version 1

Contents

1 Introduction **2**
- 1.1 Purpose of this manual . . . 2
- 1.2 The Structure of the Beg system . . . 3

2 Code Generation by Tree Pattern Matching **4**
- 2.1 Introduction . . . 4
- 2.2 Phases of the GCG . . . 8
- 2.3 More about the Description Technique . . . 8
- 2.4 Description of Addressing Modes . . . 10

3 Concepts of BEG **12**
- 3.1 Introduction . . . 12
- 3.2 Structure of a CGD . . . 12
- 3.3 Code Selection . . . 12
  - 3.3.1 Basic Structure of a Rule . . . 12
  - 3.3.2 Attributes of Operators . . . 12
  - 3.3.3 Attributes of Nonterminals . . . 13
  - 3.3.4 Shortnames . . . 13
  - 3.3.5 Conditions . . . 13
  - 3.3.6 Emit Part . . . 14
  - 3.3.7 Meaning of Rules . . . 14
  - 3.3.8 Condition Attributes . . . 14
- 3.4 Register Allocation . . . 16
  - 3.4.1 Classes of Nontermials . . . 16
  - 3.4.2 Description of Register Sets . . . 16
  - 3.4.3 Admissible Registers . . . 17
  - 3.4.4 Registers Changed by Side Effects . . . 18
  - 3.4.5 Description of Two Address Instructions . . . 18
  - 3.4.6 Spillcode and Register Copy Instructions . . . 19

4 Development of Code Generator Descriptions **20**
- 4.1 Introduction . . . 20
- 4.2 The Nonterminal Graph . . . 20
- 4.3 The Meaning of Nonterminals . . . 21
- 4.4 Correctness of a Rule . . . 22
- 4.5 Correctness of CGDs . . . 23
- 4.6 Common Usages of Nonterminals . . . 23
  - 4.6.1 Register Nonterminals . . . 23
  - 4.6.2 Addressing Mode Nonterminals . . . 24
  - 4.6.3 Nonterminals of CGD Transformations . . . 25
- 4.7 CGD Transformation . . . 25

5 The Code Generator Description Language BEGL **28**
5.1 Lexical Structure . . . 28
5.2 Structure of a CGD . . . 28
5.3 Intermediate_Code_Part . . . 28
5.4 Register Set Description . . . 29
5.5 Nonterminal Definitions . . . 30
5.6 Rule Part . . . 30
5.7 Insertions Part . . . 32
5.8 Options . . . 32

6 The GCG **33**
6.1 Structure of the GCG . . . 33
6.2 Interface to the Front End . . . 34
6.3 The Import Interface . . . 34
6.4 Insertion Points . . . 35
6.5 Options . . . 36
6.6 The Test Output Interface . . . 36
6.7 The Spill Code Interface . . . 39

7 Installation and Usage **40**
7.1 Introduction . . . 40
7.2 Files . . . 40
7.3 Usage of Beg . . . 41
7.4 Adapting BEG to Other Modula Compilers . . . 41
7.4.1 The Mocka Compiler . . . 41
7.4.2 The Sun Modula Compiler . . . 41
7.5 Structure of the Source Code . . . 41
7.6 The Dot Tool . . . 42

A Examples of Code Generator Descriptions **45**
A.1 A Simple CGD . . . 45
A.1.1 CGD . . . 45
A.1.2 Module IRCons . . . 47
A.1.3 Test Driver . . . 48
A.1.4 Normal Test Output . . . 48
A.1.5 Cover Test Output . . . 49
A.2 IBM370 CGD without folding . . . 50
A.2.1 CGD . . . 50
A.2.2 Test Driver . . . 55
A.2.3 Test Output . . . 55
A.2.4 Cover Test Output . . . 56
A.3 IBM370 CGD with folding . . . 57
A.3.1 CGD . . . 57
A.3.2 Test Output . . . 63
A.4 MC68020 CGD . . . 64
A.4.1 CGD . . . 64
A.4.2 Test Output . . . 70

1 Introduction

Automatic generation of compiler parts is known by the term "compiler–compiler". Techniques for the generation of scanners and parsers have become well known and are widely used while other compiler parts are still mainly written by hand. This manual describes a system which generates code generators. The technique of automatic generation of code generators has been used in research for several years. The code generator can be described by a relatively short *declarative* specification. Then a back end generator reads the description and automatically builds the code generator.

Developing a declarative description has a lot of advantages compared to programming the code generator by hand. First it is *easier* and much less work. So it is possible to write a description in shorter time with *less effort*. The generator can perform many consistency checks on the description and hence detect a lot of errors. Therefore an automatically produced back end is more *reliable*. Being able to quickly build reliable back ends (the main machine dependent part of a well designed compiler) allows to build *portable* compilers.

This manual describes the Back End Generator BEG. BEG makes the advantages described above available. It has been used to replace the a back end of our production Modula 2 compiler Mocka [1]. "Replace" means that the original highly tuned hand written back end was substituted by a generated one. The performance of the resulting compiler was only 10 percent slower than the original one while the code quality stayed nearly the same. We believe that this is quite acceptable even for production compilers. Therefore we think that BEG makes automatic back end generation usable in production environments.

1.1 Purpose of this manual

This manual should tell the reader how to build back ends using the BEG system. So it describes how to develop code generator descriptions and to use BEG to build back ends out of them. It does not explain how BEG works internally. This is rather interesting, however it is not necessary to know to successfully use BEG. The internal algorithms of BEG are described in [Emme88] or [ESL89].

Section 2 of this manual contains an informal introduction to code generation by tree pattern matching. Section 3 describes the concepts used by BEG. Section 4 tries to give some hints how to design a description. It also contains several examples how certain problems are solved with BEG. Section 5,6 and 7 contain more technical information, about the language BEGL, the generated code generator, the installation and use of BEG.

[1] Mocka is a trademark of GMD, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Germany West

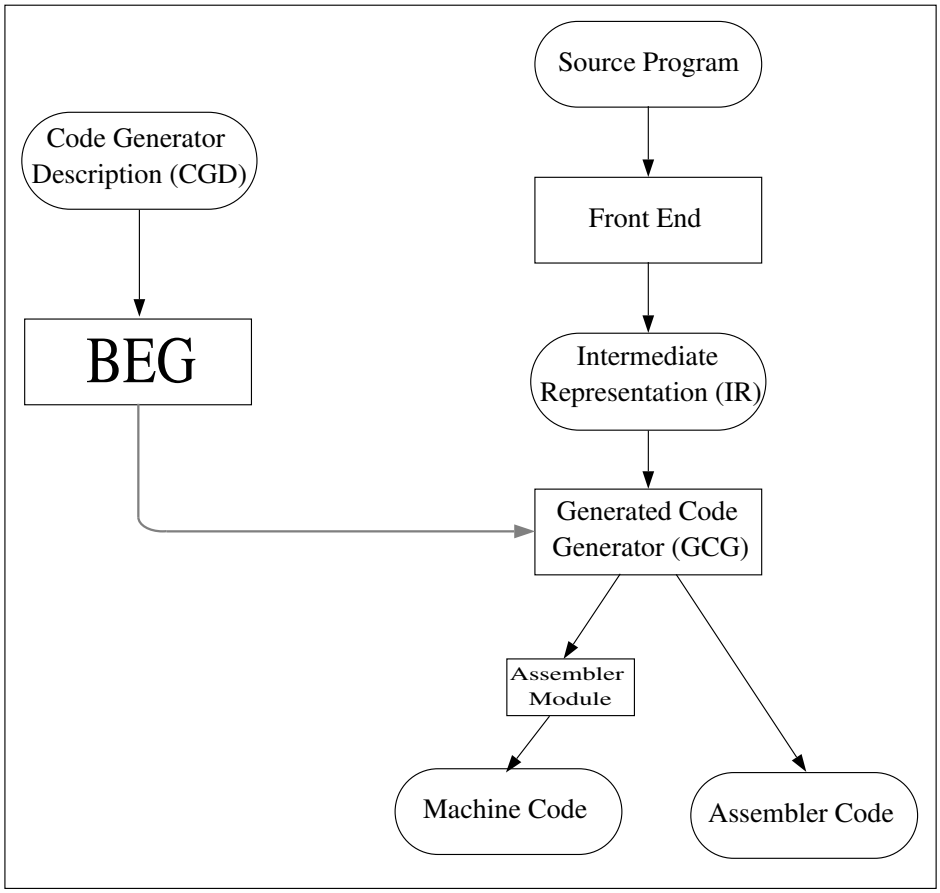

Figure 1: Structure of the Beg system

1.2 The Structure of the Beg system

Figure 1 shows the structure of a compiler using an automatically generated back end. The front end translates the source program into an *intermediate representation* (IR). Then the *generated code generator* (GCG) produces the *target code*. The BEG system builds the GCG automatically out of a code generator description (CGD). The CGD has to be written in the code generator description language BEGL. In this manual the implementation language of the GCG is just called the *implementation language*. Currently only Modula 2 is supported however a C version is planed.

BEG supports both generation of machine code and generation of assembler code. For an assembler code compiler the user has to write the CGD and some small output routines, for a machine code compiler he additionally has to provide an assembler module.

BEG supports two different register allocators the *general* and the *on the fly* register allocator. The general register allocator is slower than the on the fly register allocator but supports a wider range of target machines. Both register allocators are generated by BEG as a part of the GCG. It is also possible to use a hand written register allocator (this might be necessary for stack machines).

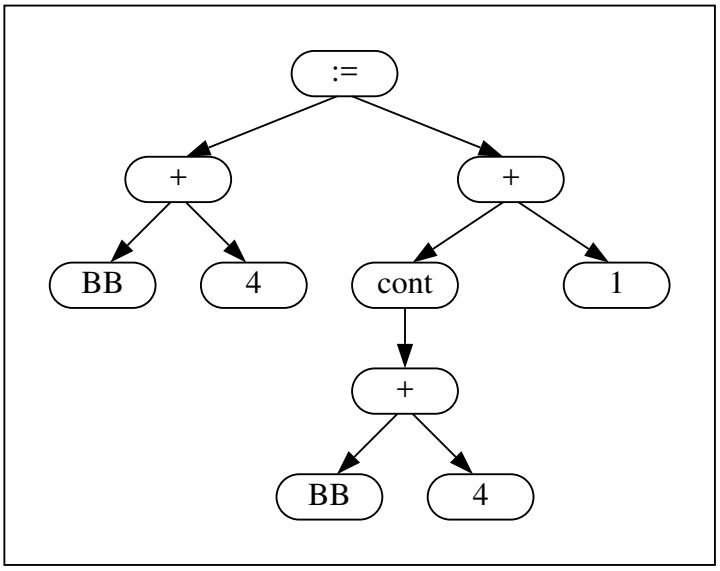

Figure 2: Sample IR expression tree

2 Code Generation by Tree Pattern Matching

2.1 Introduction

Input for the code generator is a *tree oriented* intermediate language. The program is represented as a sequence of *expression trees*. The code generator translates one expression tree at a time. So the problem the code generator solves is to produce code for a single expression tree.

The expression tree should contain all address arithmetic explicitly, e.g. addressing of an array element by using of add and multiply operators rather than a subscript operator which does the whole array addressing implicitly. However it is just a small transformation to make address arithmetic explicit.

Figure 2 contains a sample expression tree for the statement a:=a+1. a is supposed to be a local variable at offset 4 in the current activation record. So its address can be calculated by adding BB (BlockBase the starting address of the current activation record) and the offset 4. The operand of the *cont* operator is a memory address. It returns the content of the corresponding memory location. The := operator stores the value of its right operand to the memory location addressed by the left operand.

One possibility to produce code for such an expression tree is to traverse it in postfix order and to emit code for each node separately. However that results in rather bad code, figure 3 contains an example [2]

[2] A subset of IBM 370 assembler code is used throughout the manual as example language. It is particularly important to understand the concepts the examples should demonstrate, so here is a small description of IBM 370 as it is needed here: The IBM370 has 16 general purpose registers R0...R15. Instructions like A (add), S (subtract), M (Multiply) have a register form (2 address, the first register is also the result register, written AR, SR or MR) and a memory form (first operand is a register and always the result operand, the second operand is the content of a memory location). Addressing modes are

```
                        BB, block base is contained in R11
LA      R1,4            4
AR      R1,R11          +, adds R1 and R11 with result in R1
                        BB
LA      R2,4            4
AR      R2,R11          +, adds R2 and R11 with result in R2
L       R2,0(R2)        cont
LA      R3,1            1
AR      R2,R3           +
ST      R2,0(R1)        :=
```

Figure 3: Bad code produced by naive code generation

Why is this code so bad? The reason is that an usual machine instruction is so powerful that it can be used to implement *several* nodes of the tree.

The basic idea is to describe the machine instructions by *tree patterns*. Figure 4 contains the patterns we need to translate our sample expression tree. It contains code generation rules. Each rule consists of a *pattern* and the corresponding *machine instruction*.

Consider the first rule for the A–instruction. The A–instruction of the IBM 370 adds the contents of a memory location and the contents of a register. The address of the memory location is calculated as the sum of another register (here BB) and a constant (here 4). The result is returned in a register. The special problem that this register has to be equal to the operand register is not described here and will be addressed later.

It is the problem of the code generator to use these rules to translate the given input tree. The first (and most important) step is to find a *cover* of the input tree. Figure 5 shows a cover of our sample input tree of figure 2. As you can see adjacent nodes of the tree are *grouped* together. Each group of nodes corresponds to the rule whose pattern matches.

Once we have found such a cover code generation is simple. We just have to traverse the tree in postfix order and emit the instruction specified in the corresponding rule. Registers have to be allocated before, but that is discussed later. Figure 6 contains the resulting code.

So the main problem is the determination of covers. Fortunately this problem is completely solved by BEG the user just has to specify the rules. There might (and usually will) exist a lot of different covers. Each cover corresponds to a correct translation, but the code quality may vary. So we want BEG to select a cover which corresponds to good code. Therefore for each rule a cost value has to be specified. Usually one takes a weighted sum of execution time and memory requirements of the resulting instruction. The cost of

---

RS– or RX–addresses, RS–addresses are the sum of an offset (range 0...4095) and a base register (written 'offset(base register)'), RX–Addresses have an additional index register (written 'offset(base register, index register)'). Most of the instructions used here accept RX–addresses. Base or index register may be zero indicating that nothing will be added (hence R0 may not be used in addressing modes). The LA (Load Address instruction) can therefore be used to load a constant (range 0...4095) into a register. The instruction L (load) and ST (store) are used for data transfers between registers and memory.

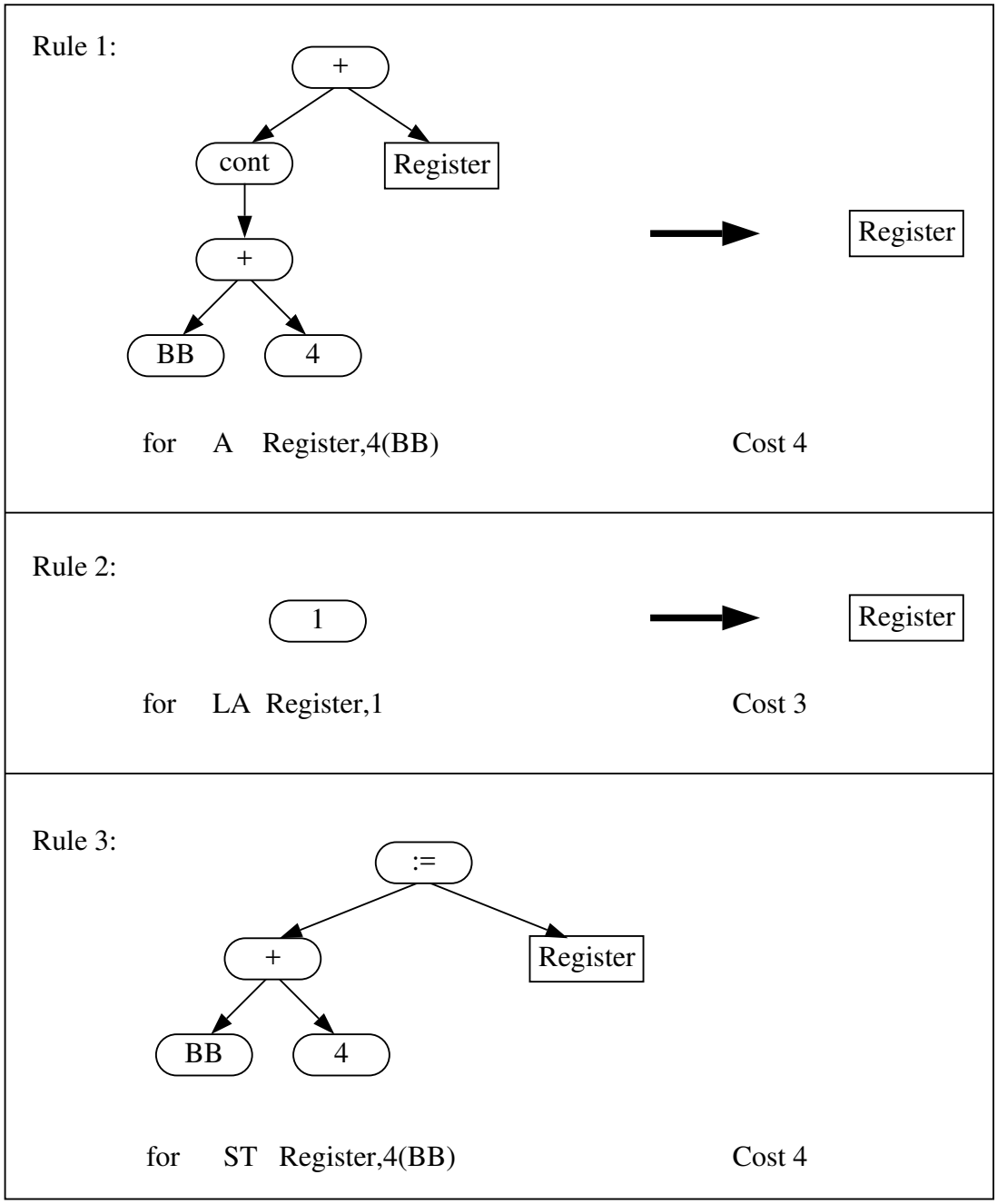

Figure 4: Rules to translate the example IR–tree

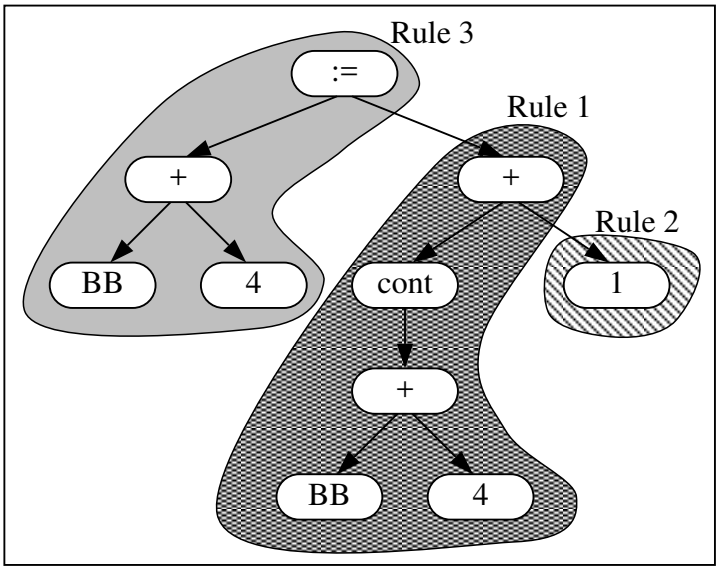

Figure 5: Cover of Sample IR tree

```
LA      R1,1
A       R1,4(R11)
ST      R1,4(R11)
```

Figure 6: Code corresponding to the cover

a cover is the sum of the costs of all rules in it. BEG manages to select a cover of minimal cost (a minimal cover) for each possible input tree.

This has some strong implications when developing a CGD:

- The user describes only a set of possible translations. BEG manages to select the optimal one according to the cost values specified by the user.
- It is not necessary for the user to know any details about the algorithm which determines these minimal covers. It is sufficient (for writing CGDs) to view BEG as a black box which efficiently determines minimal covers. Therefore the algorithm will not be described in this manual, refer [Emme88] or [ESL89] for details.
- Many complicated things can be expressed by some relatively simple rules.
- Adding of new (correct) rules can not cause any harm. Other systems which do find good covers (but not optimal ones) by some heuristics have the problem that adding of rules might cause the system to miss covers it found before.
- It is much easier to assure the correctness of a CGD than of a hand written code generator.

2.2 Phases of the GCG

Before the description method is discussed any further it is advantageous for the user to know a bit about the algorithm contained in the GCG.

The GCG is called by the front end. The tree is passed in postfix order which is usually convenient for the front end. The GCG internally builds up the expression tree and computes some information needed to determine a minimal cover. Therefore this phase of the code generator is called the *cover phase*. No code is produced until the expression tree is completely build up.

Then any algorithm can run on the tree and/or the minimal cover and calculate some attributes. For example the general register allocator does this. It traverses the tree and calculates register numbers. It is also possible (but usually not necessary) for user written algorithms to work on this tree.

Finally the *output phase* produces the target code. The tree is traversed according to the minimal cover. As explained above for each group of nodes (which correspond to a rule) the code associated with the rule is produced. More precisely, a rule might not only produce some code but additionally or instead do some attribute calculations. These attributes can be used by instructions which are emitted later.

After the output phase the expression tree is thrown away and the memory reused. So the GCG works strictly one expression tree at a time.

Summing up the GCG translates one expression in two phases: First building up the tree and the cover (cover phase), then traversing the tree according to the cover and producing code (output phase).

2.3 More about the Description Technique

Lets go back to the description problem. The description method described so far is not sufficient for real machines yet. The problem is that there are different possibilities to represent an intermediate result on the target machine. For example it could be stored in a register (as in the example above). But the machine may have different kinds of registers like data and address registers on the MC68020. Another example is that a boolean intermediate result can be stored in the condition code register, in a boolean variable, or represented by the control flow.

Therefore so called *nonterminal* symbols are introduced. A nonterminal stands for a possibility to represent an intermediate result on the machine. We call this a *storage class*. Nonterminals are written as leaves of a pattern specifying the storage classes for the operands. If the described instruction produces a result its storage class is written by '$\rightarrow$ nonterminal'. Then we call this nonterminal the *result nonterminal* of the instruction or of the corresponding rule.

In the example above we had only one nonterminal called Register. Therefore it was possible to understand it without the knowledge of nonterminals. Figure 7 gives an example using more than one nonterminal. It also contains chain rules. The pattern of a chain rule is a single nonterminal. Those rules are used to describe how an operand of a certain storage class can be transformed into another storage class. For example by copying the operand from one register class to another.

Nonterminals pose a restriction on possible covers. Suppose an instruction returns a result in storage class c_1. Another instruction uses this result. Then we have to make

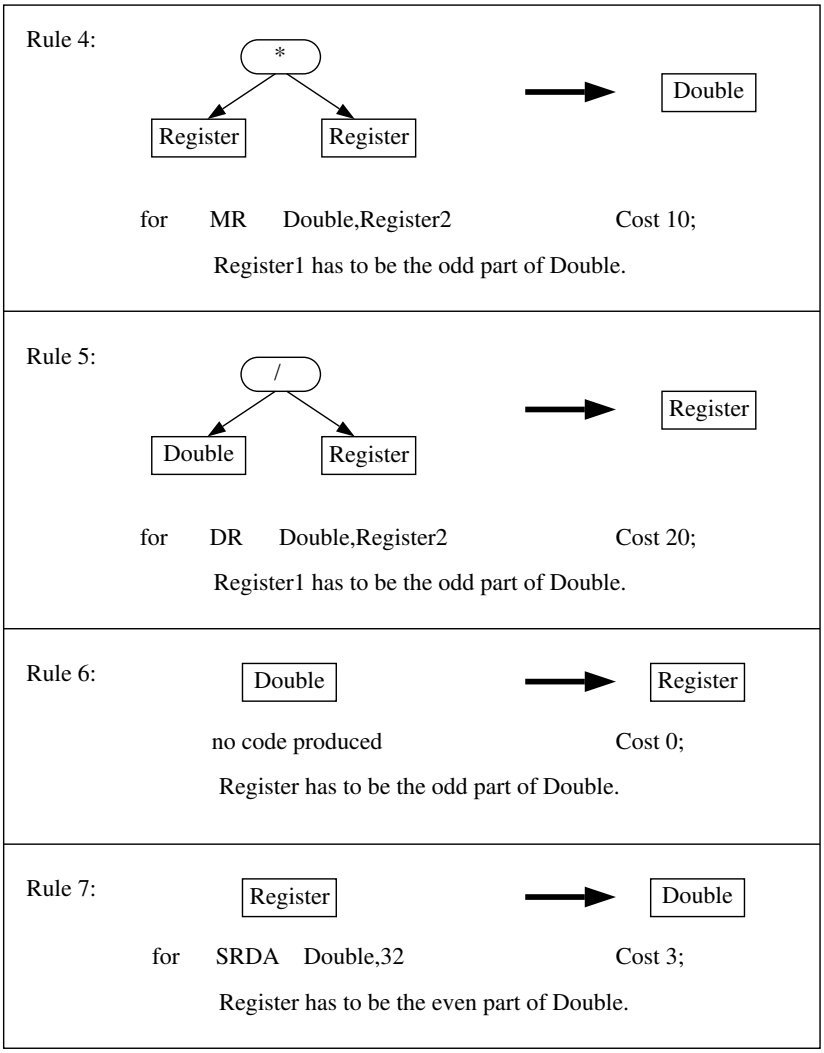

Figure 7: Rules with more than one nonterminal

This example describes the double register problem of the IBM 370. This machine has two types of registers 32–bit normal registers and double registers of 64–bit. The multiply instruction returns a double result and the first operand of the divide instruction has to be a double operand.
The main problems with this feature are in register allocation. So there are some strange constraints about register usage in the rules. However it is not important for the example to understand this completely. But if you are interested here is the complete explanation:
The double registers are not disjoint from the normal registers. In fact there are no special double registers but two normal registers can be taken together as one double register. These has to be a register with an even register number n and the register $n + 1$. So a double register has an even and an odd part which are both normal registers. Multiply and divide instructions are two address instructions. Multiply expects the first operand in the odd part of the result register and the divide instructions returns the result in the odd part of the first operand register.

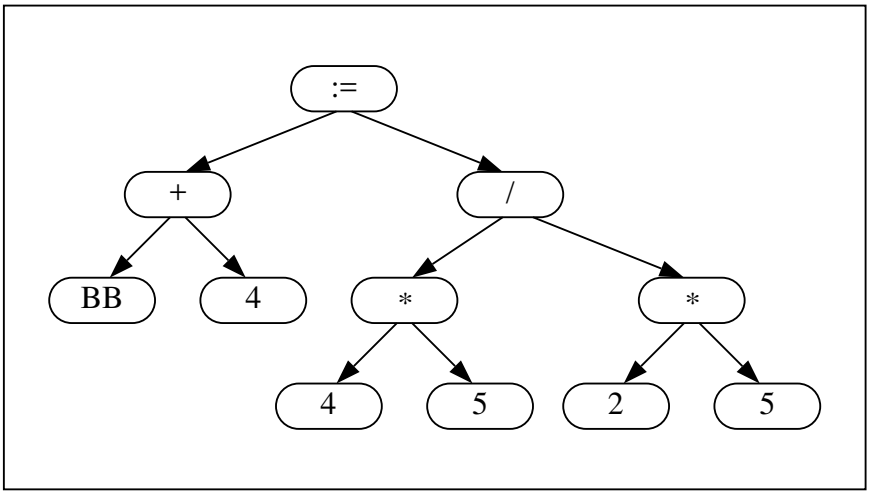

Figure 8: Sample expression tree

sure, that this second instruction expects its corresponding operand also in storage class c_1. In terms of covers this means the following: Suppose a rule matches somewhere in the input tree. Let n be a nonterminal contained in the pattern of the rule. This nonterminal matches against a certain node of the tree. This node has to be covered by a rule with nonterminal n.

Figure 8 contains a sample expression. Figure 9 shows a cover of this input tree according to the rules given in figure 4 and figure 7. Each arrow leaving one group of nodes and entering another can be labeled with a nonterminal: Register or Double. This represents the storage class of the corresponding intermediate result. Look at the application of Rule 6. Because it is a chain rule it does not cover any nodes of the expression tree. It merely transforms an intermediate result of storage class register into one of storage class double. So it changes the label of the arrow and emits the corresponding target code. Figure 10 contains the resulting target code. The correct register numbers are obtained by the register allocator. There is a complete code generator description handling this problem contained in the appendix.

2.4 Description of Addressing Modes

Nonterminals are also used to describe addressing modes. That might be an even more important application for nonterminals as the things described above.

Consider the way we have described the A instruction or better one addressing mode of the A instruction. If we extend this to all addressing modes we might get n rules. When trying to describe other instructions with the same method, we had to write n rules for each instruction. This leads to $n * m$ rules where m is the number of instructions. So this method leads to impractical rule numbers.

Instead we introduce a nonterminal for the addressing mode. For example RXAddress. We write rules for each possible form of RX–Addresses leading to the nonterminal RXAddress. Then we have to write only one rule for each instruction which uses an RX–Address.

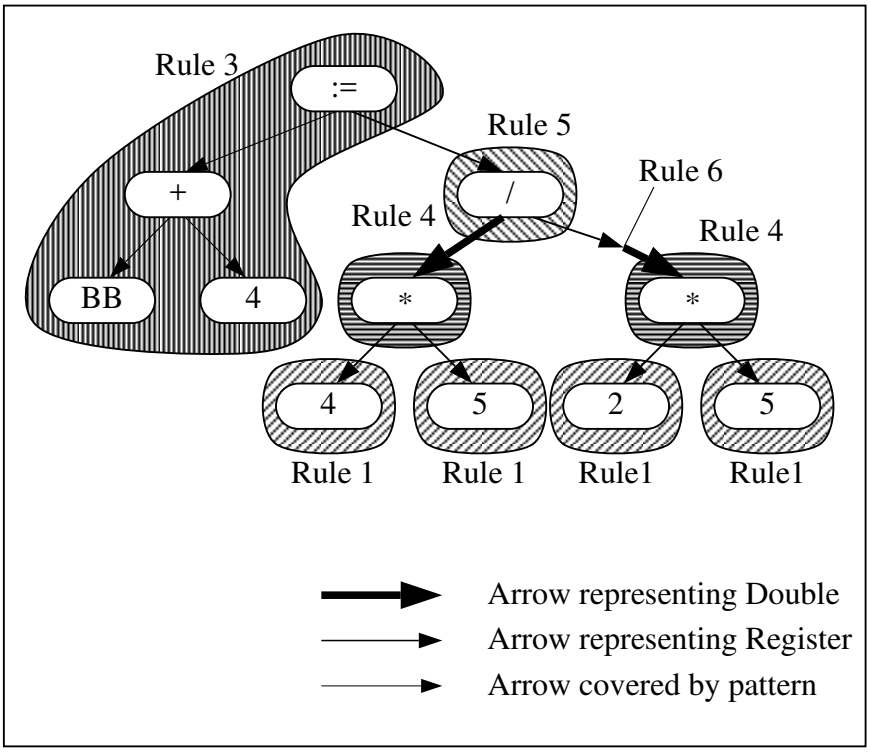

Figure 9: Cover of the expression tree

```
LA      R3,4
LA      R0,5
MR      D2,R0
LA      R1,2
LA      R0,5
MR      D0,R0
DR      D2,R1
ST      R3,4(R11)
```

Figure 10: Code corresponding to the cover

So we get $n + m$ rules instead of $n * m$. Section 4 contains some more details about this. There are also some examples.

3 Concepts of BEG

3.1 Introduction

The last chapter explained the basic ideas behind the method of code generation by tree pattern matching. This section describes how these concepts are used with BEG. The concepts behind the language BEGL are explained. The concrete syntax of BEGL is described in section 5.

This section is split into two parts one about code selection the other about register allocation. Though these two concepts seem to be mixed in BEGL (because the aspects of code selection and register allocation of one instruction are described together) it is possible to understand code selection without any knowledge of register allocation.

There are a lot of example CGDs in the appendix. It might be useful to look at them (or even play around with them) while reading the next three chapters. The first example CGD was designed to demonstrate the concepts of BEG and their usage while the other CGDs should demonstrate the proper design of CGDs.

3.2 Structure of a CGD

A CGD mainly consists of rules as described above. They are contained in the *rule part* of the CGD. There are also some other shorter parts. The intermediate representation is described in the *intermediate code part*. Basically the operators are enumerated and their arity is specified. In the *register set part* the register set of the target machine is specified. The nonterminals are defined in the *nonterminals part*. Then follows the *rules part*. At last there is the *insertions part* which allows to customize the generated code as desired.

3.3 Code Selection

3.3.1 Basic Structure of a Rule

A rule in the rule part has the following basic structure:

```
RULE pattern [→ result_nonterminal]
    COST Integer;
    EMIT Instruction;
```

The pattern is written in prefix notation. This is possible as the arities of the operators and the names of the nonterminals are known. In BEGL the Instruction in the *emit part* is just a piece of implementation language code. It usually contains statements to output the desired instruction. Though this sometimes looks not very nice in a CGD it is a very flexible concept. In the examples below however we will just write the assembler instruction which should be produced.

3.3.2 Attributes of Operators

An attribute of an operator stores a value known at compile time. The front end computes these attribute values and passes them to the back end. Then the attributes might be used in the emit part of rules to produce the correct machine instruction. An example is the IntegerConstant operator. It usually has an attribute value of type Integer.

```
RULE IntegerConstant → Register;
    COST 4;
    EMIT LA Register.register,IntegerConstant.value
```

The value of the attribute can be accessed by writing the operator identifier followed by a dot and the attribute name. The term 'Register.register' is described later. The attributes for each operator have to be defined in the intermediate code part of the CGD. They can have any type allowed for record components in the implementation language. These can be predefined or user defined types.

3.3.3 Attributes of Nonterminals

Nonterminals may also have attributes. They are used to compute synthesized attributes during the output phase. Their values are calculated by the code contained in the emit part and might be used in the emit part of other rules.

Nonterminals corresponding to registers have a special attribute called register. It is computed by the register allocator and has to be used to insert the correct register numbers into the instruction. As an abbreviation the term '. register' can be left out.

The next example shows how to express constant folding in a CGD using attributes of nonterminals. Therefore we need a nonterminal called Constant with an attribute value:

```
RULE IntegerConstant → Constant;
    COST 4;
    EMIT Constant.value := IntegerConstant.Value

RULE IntegerPlus Constant.a Constant.b → Constant;
    COST 4;
    EMIT Constant.value := a.value + b.value;
```

The last rule needs to use short names which are described in the next section.

3.3.4 Shortnames

It might happen that a nonterminal or a operator occurs more than once in the pattern. So the normal attribute access would be ambiguous. Therefore the nonterminal or operator name can be renamed by appending a dot followed by a short name. Then for attribute accesses the short name has to be used. The scope of a short name is one rule.

An example is the rule for constant folding above. The pattern contains the nonterminal Constant twice. So the first one is renamed with the short name a and the second one with the shortname b. It is also possible to rename the result nonterminal, however this was not necessary.

3.3.5 Conditions

Sometimes it is necessary to restrict the applicability of a rule according to some attributes of the intermediate representation. Therefore a rule might contain a condition. The rule can only be applied if the condition yields true. The condition has to be a correct condition of the implementation language. Example:

```
RULE IntegerConstant → Register;
    CONDITION (IntegerConstant.value>=0) AND
        (IntegerConstant.value<=4095)
    COST 4;
    EMIT LA Register.register,IntegerConstant.value
```

Conditions may access attributes of operators but they must not access the attributes of nonterminals. The reason of this restriction it that BEG needs to evaluate the conditions in the cover phase while the attributes of nonterminals are calculated in the output phase. This restriction is relaxed a bit in section 3.3.8.

3.3.6 Emit Part

The emit part contains implementation language code. BEG performs some text substitutions in this code to make attribute accesses possible. Each shortname eventually followed by an attribute selection is replaced by a corresponding implementation language expression. If shortnames conflict with identifiers of the implementation language the shortname has to be changed.

3.3.7 Meaning of Rules

For a given input tree Beg determines a minimal cover according to the specified rules. Only rules whose condition yields true are applied therefore. Afterwards the cover tree is traversed. Sons are processed from left to right. For each node the code specified in the emit part is executed.

3.3.8 Condition Attributes

Condition attributes are also attributes of nonterminals but work differently than those described before. They may be accessed within conditions while the others may not.

Condition attributes are a quite powerful concept if used wisely. The problem is, that their usage can affect the optimality of BEG. That means if those attributes are used in the wrong way BEG will not find a minimal cover under all conditions. Fortunately condition attributes are really necessary only in rare cases. However they are convenient and can be used in a way not effecting minimality. So they have been included into BEG.

Condition attributes are evaluated during the cover phase rather than the output phase. Therefore attribute calculations have to be specified in the EVAL part rather than the EMIT part of the rules. Their values can be accessed in Conditions.

The main application is constant folding. Folding can also be done with normal attributes, however then the result of the folding can not be checked in conditions. This is necessary in some cases, for example some addressing modes are applicable only if their offset is in a certain range.

```
RULE IntegerConstant → Constant;
    COST 4;
    EVAL Constant.value := IntegerConstant.Value
```

```
RULE IntegerPlus Constant.a Constant.b → Constant;
    COST 4;
    EVAL Constant.value := a.value + b.value;

RULE ...Constant ...;
    CONDITION ...Constant.value ...
```

There are certain conditions which allow to use condition attributes and keep minimality. For each nonterminal N which has condition attributes one of the following constraints must hold:

- For a certain tree node there can only match one rule with result nonterminal N.
- More than one rule can match but one is so good (leads to such a low cost value) that all other matching rules can not possibly be contained in a minimal cover [3].

The first rule is typically fulfilled by the folding rules. In the appendix there is a CGD with the nonterminal AregDispl or AbsRx and AbsRs. For those the second condition is fulfilled. However BEG currently does not check these conditions automatically.

[3] In fact the following is sufficient: If we have a minimal cover containing a bad rule it is always possible to find a minimal cover using the good rule.

3.4 Register Allocation

3.4.1 Classes of Nontermials

BEG can generate register allocators for a variety of target machines. Most of the information needed by the register allocator are already contained in the rules of a CGD. However some more information is necessary about the instructions produced by the emit parts. The register allocator basically distinguishes between three different kinds of rules:

- *Register rules:* The rule emits one (or more) machine instruction(s). It uses some values contained in operand registers and produces a result in a result register. After the instruction the operand registers are free again and may be used for other purposes.
- *Addressing mode rules:* The rule does not emit any instructions. It just calculates some attributes which are used later. This is typically the case for rules building up an addressing mode. Such a rule might have some operands contained in registers. It will for example use the register numbers to construct the addressing mode for another instruction. However the contents of these registers are not used by the code produced by this rule and so the registers are not free again. That is very important for the register allocator to know.
- *Memory rules:* This kind of rules is nearly the same as the first kind. However the result is not put into a register it is put somewhere else perhaps on a stack or in memory. So the register allocator need not select a register for the result.

If you look a bit closer at these properties you might notice that these strongly relate to the storage class or nonterminal of the result. So the kind of the rule is not specified on a per rule base but it is specified per result nonterminal. The nonterminals are partitioned into these tree classes: register nonterminals, addressing mode nonterminals and memory nonterminals. BEG determines the class of the rule out of the class of the result nonterminal.

So the user has to specify the class of each nonterminal. Out of these information BEG deduces the kind of each rule. It then assumes that the rule behaves in the way described above depending on its kind. BEG can not check this because the emit part is implementation code which is not analyzed by BEG. So it very important that the rules behave like BEG expects them to do. If they do not this might result in a wrong register allocation. However it is not very difficult to assure this and then you will get a reliable register allocator.

3.4.2 Description of Register Sets

To generate a register allocator BEG needs to know about the register set of the target machine. The register set is described in the register set part of the CGD. The registers of the target machine have to be enumerated. Example the register set of the MC68000:

D0,D1,D2,D3,D4,D5,D6,D7,A0,A1,A2,A3,A4,A5,A6

There are some target machines with registers which are parts of other registers. For example the AL register is a part of the AX register on Intel 8088 processors. The double

register problem of the IBM 370 is just the same thing: There is a double register which consists of two normal registers. Therefore a *part relation* may be specified. For each register the registers it contains can be enumerated.

BEG computes the *disjoint relation* out of the part relation. Two registers are disjoint iff they do not have a common part. If two registers are not disjoint this means for BEG that assigning a value to one of them will destroy the value contained in the other.
Example the register set of the IBM370, the registers in parentheses are defined as part of the register in front of them:

```
R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,
D0(R0,R1),D2(R2,R3),D4(R4,R5),D6(R6,R7),D8(R8,R9),
D10(R10,R11), D12(R12,R13), D14(R14,R15),
F0, F1, F2, F3, F4, F5, F6, F7,
DF0 (F0,F1), DF2(F2,F3), DF4(F4,F5), DF6(F6,F7);
```

3.4.3 Admissible Registers

For each register operand and the result of an instruction the *admissible registers* can be specified. A register operand means an operand contained in a register. Such an operand is represented by a register nonterminal contained in the pattern. Admissible registers for the result can only be specified if the instruction produces a result in a register, that means if the result nonterminal is a register nonterminal.

The register allocator will pick one of the admissible registers for each operand and for the result (if the instruction produces a result in register). The result of the register allocation is the attribute called register. Each register nonterminal has this attribute. It is guaranteed that the value of this attribute is contained in the set of admissible registers.

To shorten the CGD for each register nonterminal a default register set has to be specified. So if register set is specified the corresponding default register set is taken. In practice there are only few rules where the admissible registers have to be specified explicitly. Example for RS addresses on IBM 370. The Register R0 may not be used for addressing:

```
RULE Register (R1..R15) → RSAddress;
    COST 0;
    EMIT RSAddress.basereg := Register.register;
         RSAddress.offset := 0;
```

Example with sets of admissible registers containing only one register. The operator IntegerPower is implemented by calling a routine of the runtime system which expects its first operand in R1 the second in R2 and returns the result in R1.

```
RULE IntegerPower Register (R1) Register (R2) → Register(R1);
    COST 20;
    EMIT BAL R14,IntPower
```

When assigning registers BEG might take one of the operand register as the result register. This is usually allowed for instructions. However care has to be taken if a sequence of instructions is emitted. Sometimes it is convenient to use an operand register to hold a

temporary result. That means that the instruction changes an operand register as a side effect. This is allowed *currently* because BEG does not eliminate common subexpressions at the moment. However it will be not in future versions which do common subexpression elimination and therefore might use the value twice.

3.4.4 Registers Changed by Side Effects

Some instructions change the contents of one (or more) registers as side effect. You can tell the register allocator by enumerating the changed registers in the rule. Example:

```
RULE IntegerPower Register (R1) Register (R2) → Register(R1);
    COST 20;
    CHANGE (R14);
    EMIT BAL R14,IntPower
```

Currently BEG has no language construct which allows to get a temporary register out of a certain set of registers needed by the code produced by the emit part. However it is possible to simply use a fixed register and specify this fact using CHANGE. When using the on the fly register allocation CHANGE is implemented by spilling the changed registers which are in use. The general register allocator will take all change clauses during the life time of an intermediate result into account when picking a register.

3.4.5 Description of Two Address Instructions

Two address instructions have the additional constraint that the result register and the register of one operand have to be equal. There is a special language construct which allows to declare one operand as the target operand. In the simplest case the set of admissible registers of the target operand and that of the result operand are equal. Then BEG will assign the same register for both. Example:

```
RULE IntegerPlus Register.a Register.b → Register;
    COST 2;
    TARGET a;
    EMIT AR a.register,a.register
```

If result and target operand have different admissible registers at least the following condition must hold. For each register contained in one set there has to be a register in the other set in a way that both are not disjoint. When selecting registers BEG guarantees that the register assigned to the result and the register assigned to the target operand are not disjoint. That means they have to have a common part which contains the value. This complex feature can be used to handle the double register problem (see CGD example in the appendix). Example how to truncate a double into a single register by taking the odd part of the double register:

```
RULE Double (D0,D2,D3) → Register (R1,R3,R5);
    COST 0;
    TARGET Double;
```

3.4.6 Spillcode and Register Copy Instructions

The register allocator might insert register copy instructions if necessary. For example if the admissible registers of a result and of the operand where the register is used are disjoint.

In fact the register allocator can not insert instructions itself but it calls a user routine which has to emit the register copy instruction.

However sometimes register sizes or types are different and therefore the semantic of a register copy would be undefined. For example it would be wrong to copy double float value temporarily to an integer register. So BEG guarantees that a value is kept only in the registers admissible for the result of the rule which produced this value or admissible for the operand which uses this value. Therefore it is guaranteed that no bad register copies are requested.

Only the general register allocator can produce spill code in every situation. It expects a stack to be used. Therefore the user has to provide two routines, one to push the contents of a certain register on the stack the other to pop the contents from the stack. Of course a stack can be simulated easily if desired. BEG spills out of an admissible register for the result and reloads into an admissible register for the operand.

The on the fly register allocator does only spill registers changed by side effects. Currently it will block if an expression is to complex.

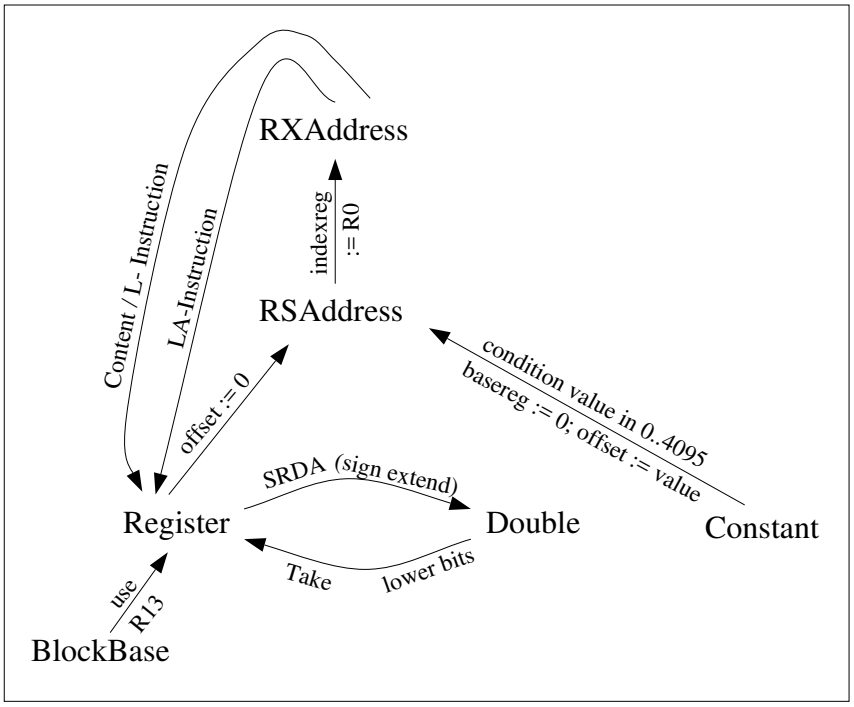

Figure 11: Nonterminal graph of CGD in A.1.2

4 Development of Code Generator Descriptions

4.1 Introduction

Development of CGDs is not a very difficult job but it requires another way of thinking not procedural but declarative. Therefore this section contains several methods to visualize what is going on and several examples.

The first step when designing a CGD is the definition of the nonterminals and the most important chain rules. You have to do this first because without knowing the exact meaning of the nonterminals you can not specify any instruction. On the other hand the definition of the nonterminals is very important and the most difficult part when designing a CGD. A wrong definition of nonterminals can make it impossible to describe a certain instruction correctly. It also can also force you to write much more rules than really necessary. So the decision about the nonterminals should be made very carefully.

There are two goals when defining the nonterminals. First it should be possible to describe every instruction completely. Second this should be possible with as few as possible rules. For the second goal there exist some CGD transformations which are quite helpful.

Therefore the main part of this sections addresses the problem how to select the right nonterminals.

4.2 The Nonterminal Graph

The kernel of a CGD is the definition of the nonterminals and the chain productions.

When nonterminals and chain rules have been selected well everything else is straight forward. The *nonterminal graph* is a method to make the dependence of nonterminals and chain rules visible.

The nodes of the nonterminal graph are the nonterminals of the CGD. There is an edge between two nonterminals n_1 and n_2 if there is a chain rule $n_1 \rightarrow n_2$. Often it is useful to write the cost and the action of the chain rule at the edge when drawing the graph.

Sometimes it is also useful to draw edges for some selected non chain rules. That is only possible if there is only one nonterminal in the pattern of the rule. That means operators have arity less or equal 1. Then the edge is labeled with the pattern, the cost and the action. Figure 11 contains a sample nonterminal graph.

4.3 The Meaning of Nonterminals

A nonterminal can be interpreted as a storage class, a way how an intermediate result is implemented on the target machine. Theoretically this is a function of the current machine state. It returns the current value of the intermediate result. This function is called *characteristic function* ϕ_N of a nonterminal N. ϕ_N is parameterized by the attributes of the nonterminal so it is actually a function of machine state and attributes yielding the intermediate result.

When defining a nonterminal the characteristic function should be clear and be included as a comment in the CGD. Here are some examples:

Register: The intermediate result is stored in a register. Attribute is the register number r. The characteristic function is the contents of register r or $\phi_{\text{Register}} = R_r$.

NegRegister: Instead of the value itself the negative value is stored. The characteristic function is $\phi_{\text{NegRegister}} = -R_r$.

RegDispl: Attribute is the register number r and an offset d. The value of the intermediate result minus d is stored in register r. So the characteristic function is $\phi_{\text{RegDispl}} = R_r + d$.

Constant: Attribute is a value v. The characteristic function is $\phi_{\text{Constant}} = v$.

RXAddress: Attributes are the number of an index register i and of a base register b and an offset d. The nonterminal should represent the behavior of the RX addressing mode of the IBM 370. So the characteristic function is
$\phi_{\text{RXAddress}} = d + R_i + R_b$ if $i, b \neq 0$;
$\phi_{\text{RXAddress}} = d + R_b$ if $b \neq 0, i = 0$;
$\phi_{\text{RXAddress}} = d + R_i$ if $i \neq 0, b = 0$;
$\phi_{\text{RXAddress}} = d$ if $i, b = 0$.

ContRX: It has the same attributes as RXAddress. The value is not the address but the value stored at this address in memory: $\phi_{\text{ContRX}} = \text{Memory}[\phi_{\text{RXAddress}}]$.

These definitions directly lead to some (chain) rules. For example

```
Register → NegRegister;          EMIT Negate instruction;
Register → RegDispl;             RegDispl.d := 0;
RegDispl → RXAddress;            RXAddress.d := RegDispl.d
                                 RXAddress.b := RegDispl.r
                                 RXAddress.i := 0
Content RXAddress → MemRX; do nothing
```

4.4 Correctness of a Rule

In this section we assume that the intermediate representation has no side effects, i.e. only top level operators change the machine state, all other operators behave like functions [4].

A rule means that whenever it is applicable, i.e. the pattern matches in the input tree, the condition holds and the subtrees matching against the nonterminals in the pattern are covered accordingly, then the operators contained in the pattern can be translated into the code specified by the EMIT part.

The user has to make sure that this condition holds for every rule in the CGD. However this can be checked locally by just considering the rule and the characteristic functions of nonterminals occurring in it. Then the BEG approach assures that only correct code is produced.

Now we want to describe the phrase 'can be translated into' more precisely. We concentrate only on non top level rules. According to the operators in the pattern and the definition of the intermediate representation the code to produce has to compute a certain function f. The arity of f is the number of nonterminals in the pattern. f is parameterized by the attributes of the intermediate operators. f might depend on the state of the target machine, for example the Content operator.

Let m be a machine state and m' the state after execution of the instructions produced by the emit part of the rule. Then in the state m' the function f must have been computed. That means $\phi_R(m', a_R)$ has to be the correct result where R is the result nonterminal and a_R are the attributes of R. The operands of f are $\phi_N(m, a_N)$ where N is a nonterminal contained in the pattern and a_N are its attributes. Hence the following equation must hold:

$$f(\phi_{N_1}(m, a_{N_1}) \ldots \phi_{N_i}(m, a_{N_i}), m) = \phi_R(m', a_R)$$

For example consider the rule

```
RULE Plus Register.a Register.b → Register.r;
    COST 4;
    TARGET a;
    EMIT AR r.register, b.register
```

Obviously $f(a, b, m) = a + b$. The last parameter is the machine state and not used here. $\phi_{Register}(m, a) = m.R_a$ where the attribute a is the register number. $m.R_a$ means the register with number a within machine state m. This definition is in principle the same as the definition in the section before. So we get the equation

$$f(\phi_{Register}(m, a.\text{register}), \phi_{Register}(m, b.\text{register}), m) = \phi_{Register}(m', r.\text{register})$$

$$= f(m.R_{a.\text{register}}, m.R_{b.\text{register}}, m) = m'.R_{r.\text{register}}$$

[4] BEG can work very well with operators with side effects. Only the theory becomes more complicated.

$$= m.R_{a.\text{register}} + m.R_{b.\text{register}} = m'.R_{r.\text{register}}$$

Because of the TARGET specification in the rule is a.register = r.register and hence we get

$$m.R_{r.\text{register}} + m.R_{b.\text{register}} = m'.R_{r.\text{register}}$$

which is the description of the AR instruction.

The equation described before is necessary for the correctness of a rule but it is not sufficient. It only guarantees that the code produces the correct result. Additionally the code may not 'destroy' other parts of the machine state. However to describe this more formally is beyond the scope of this manual.

4.5 Correctness of CGDs

A CGD is said to be partial correct if it produces correct code for each input statement it can process. A CGD is complete if it can process every possible input tree. So a CGD is correct if it has these both properties. Now the question is how to assure the completeness of a CGD.

Because of the optimality of the BEG approach CGDs have the following property. Adding new rules to a complete CGD always leads to complete CGDs. So one can start with a simple CGD prove its correctness and afterwards extend it to produce good code. However proving completeness is usually so easy that it also works with big CGDs.

CGDs normally have the following nice property: For each expression tree code can be produced which computes the result in a register. Usually this can be proved easily by induction over the tree. If this is true it simply has to be checked if there are rules for each top level operator accepting operands in registers. If so completeness has been proved.

4.6 Common Usages of Nonterminals

4.6.1 Register Nonterminals

A normal CGD has at least one register nonterminal [5]. It can have more to guide register allocation. In fact similar things can be described with one register nonterminal putting the whole problem on the register allocator or by using more nonterminals.

One register nonterminal is a good choice to keep the CGD simple. The register allocator is quite powerful so it usually produces good results. However it is useless to describe two instructions with the same patterns and conditions but with different register requirements and different costs. In that case instruction selection always takes the cheapest instruction. It can not take into account that registers are a limited resource and that it might be better to take the more expensive instruction to prevent spilling. The register allocator is always bound to the instructions selected before and will meet their register requirements.

However this is a rare problem. There are different possibilities to solve the problem: More register nonterminals can be used to guide code selection. For example on the Motorla processor it might be useful to use address registers for integer arithmetic (to be able to use the lea instruction). Then two nonterminals one for data and one for address

[5] You can have no register nonterminals to switch off the register allocators of BEG

registers can be introduced. The different instructions can then described without problems. The advantage of that method is that the instruction selection minimizes coercion and operator costs. However it assumes that an infinite number of registers is available.

That only becomes a problem if register classes are very small. For example on Intel processors exist some instructions which work with every register, but are faster and shorter when the accumulator register AX is used. Introducing a nonterminal for the accumulator is of no help. Still instruction selection would use only the instructions with the AX register perhaps forcing the register allocator to introduce register copy or spilling instructions. However there is a much simpler trick to describe this. When selecting registers the register allocator prefers registers specified earlier in the register definition. So just define the AX register first and the register allocator will use it most often.

If the target machine is simple enough the fast on the fly register allocator can be used instead of the general one. However on the fly produces bad register allocations if the machine is too complex. Sometimes it is possible to guide the on the fly register allocator by introduction of new nonterminals. This allows to handle a bit more complex register sets.

4.6.2 Addressing Mode Nonterminals

In spite of the name there is not one addressing mode nonterminal per addressing mode. There is one nonterminal per group of addressing modes possible in an instruction. On the MC680x0 there is only one such nonterminal (named ea) because every possible addressing mode is allowed in each instruction.

On IBM 370 there are two kinds of instructions those accepting only RS Addresses (register + offset) and those accepting RX Addresses (base register + index register + offset) and RS Addresses. So there are two nonterminals one representing RS Addresses and one representing both RX and RS Addresses.

The term addressing mode is used with a slightly different meaning by different machine definitions. An addressing mode first can be just a function which computes a memory address. An instruction using such an addressing mode accesses (reads or writes) this memory location or simply stores the address (like the lea or LA instructions). Those addressing modes can be described in the following way:

```
RULE AddressPlus Register Constant → RSAddress;
    COST 0;
    EMIT RSAddress.offset := Constant.value;
         RSAddress.baseregister := Register.register;
RULE Register → RSAddress;
    COST 0;
    EMIT RSAddress.offset := 0;
         RSAddress.baseregister := Register.register;
RULE RSAddress → Register;
    COST 3;
    EMIT LA   Register,RSAddress.offset(RSAddress.baseregister);
RULE Content RSAddress → Register;
    COST 4;
    EMIT L    Register,RSAddress.offset(RSAddress.baseregister);
```

```
RULE Plus Register Content RSAddress → Register;
    COST 4;
    EMIT A    Register,RSAddress.offset(RSAddress.baseregister);
```

This works pretty well. However there are other addressing modes on the MC68020 which do not fit into this scheme. Here an addressing mode can be everything which can be an operand of an instruction. For example the content of a register or an immediate value. A lea instruction applied on such an addressing mode is meaningless. There is also no content in the tree when an instruction uses the value of such an operand. Those operands can be described by rules like that:

```
RULE Plus Register Constant → Register;
    COST 4;
    EMIT add    register,#Constant.value;
```

However because these addressing modes can be combined freely with the other ones and because we do not want to write more than one rule to describe this aspect of each machine instruction we introduce a new nonterminal named ea. Then the MC68020 can be described as follows (see the appendix for the complete rules).

```
RULE AddressPlus Register Constant → Dest;
RULE Register → Dest;
    EMIT build addressing mode register indirect
    ...all addressing modes
RULE Content Dest → ea;
RULE Constant → ea;
RULE Register → ea;
    EMIT build addressing mode register direct
RULE Dest → Register;
    EMIT lea instruction
RULE ea → Register;
    EMIT mov instruction
RULE Plus Register ea → Register;
```

4.6.3 Nonterminals of CGD Transformations

The CGD transformations described below can introduce some new nonterminals. See below.

4.7 CGD Transformation

There are some possibilities to transform a CGD without changing its meaning. The goal of these transformations is to keep the number of rules small. Usually nobody will perform such a transformation formally as described here. Normally those transformations are done intuitively. However studying them gives a better understanding.

Figure 12 shows the most important transformation rule. A complex pattern is split into two parts. A new nonterminal N and a rule which derives N out of the subpattern t is introduced. Then in the initial complex pattern the subpattern t can be replaced by N.

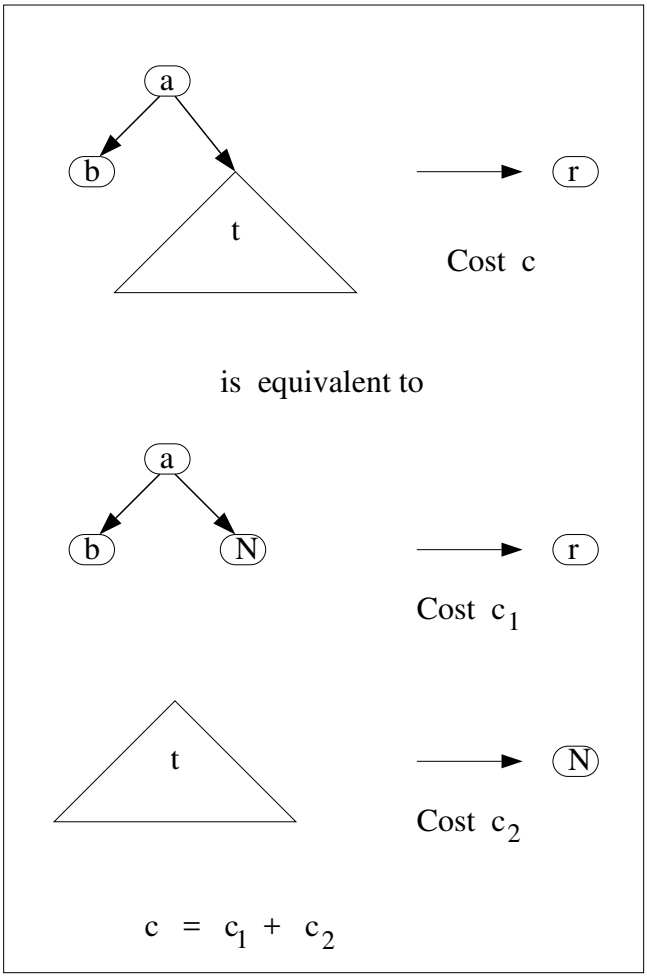

Figure 12: Rule Transformation

The transformation keeps the meaning of the CGD, it can be applied without doing any harm (except increasing the number of nonterminals, which is bad for readability and efficiency). However a smaller pattern limits the attributes visible in conditions and emit parts and these might have to be changed. That is always possible to overcome this problem by introducing new nonterminal attributes. For attributes used in conditions of course condition attributes have to be used. However because of the transformation this does not lead to any problems with the optimality.

It often happens that a subpattern occurs rather often. Then the transformation rule can be used to factor it out. After that transformation the subpattern occurs only once and all other occurrences are replaced by a nonterminal.

After this transformation it is sometimes possible to combine several of the resulting rules. In fact that is what has been done to describe the addressing modes and which lead from a quadratic number of rules to a linear.

Theoretically this transformation can be applied to all rules with patterns containing

more than one operator. This process leads to an equivalent CGD which contains only simple patterns (patterns containing at most one operator). So it is in fact possible to describe everything which can be described by complex patterns also with simple patterns.

5 The Code Generator Description Language BEGL

5.1 Lexical Structure

Comments, identifiers (Ident) and integer numbers (Integer) are defined as in Modula 2. Additionally identifiers may contain the underline character. All characters of an identifier are significant, upper and lower case letters are distinguished.

A CGD may contain implementation language text (Target_Text) written in curly brackets { }. It can extend over multiple lines. It may contain every character allowed in the implementation language, however curly brackets have to occur in pairs or have to be contained in string constants of the implementation language.

EBNF is used to describe the syntax of the language. Angular brackets [] denote optionality and curly brackets { } repetition (possibly 0 times). The construction $\{a//b\}$ is an abbreviation for $a\{ba\}$. Nonterminals are written as English words containing lower case letters. Reserved words of the language are written using bold capital letters. Special symbols of the language are quoted.

5.2 Structure of a CGD

CGD ::=**CODE_GENERATOR_DESCRIPTION** Ident ';'
Intermediate_Code_Part
Register_Set_Part
Nonterminals_Part
Rules_Part
[Insertions_Part]
END CODE_GENERATOR_DESCRIPTION
Ident '.'.

The identifiers following the word 'CODE_GENERATION_DESCRIPTION' have to be equal. They are used to name a particular description. The name is also used to name the interface module of the back end.

5.3 Intermediate_Code_Part

Intermediate_Code_Part ::=**INTERMEDIATE_REPRESENTATION**
NONTERMINALS
{ Inter_Type_Ident // ',' } ';'.
OPERATORS
Operator_Definitions.

Inter_Type_Ident ::=Ident.

Operator_Definitions ::=Operator_Ident [Attribute_Definitions]
[Operands]
['->' [Inter_Type_Ident]] '.'

Attribute_Definitions ::='(' { Attribute_Ident ':' Type // ';'} ')' .

Operands ::=Inter_Type_Ident '+' Inter_Type_Ident
| Inter_Type_Ident { '*' Inter_Type_Ident }.

Type ::=Modul_Ident '.' Type_Ident | Type_Ident.

Operator_Ident, Inter_Type_Ident, Attribute_Ident, Modul_Ident,
Type_Ident ::=Ident.

Example:

```
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant  ( v : INTEGER )                -> Value;
  Plus                        Value + Value -> Value;
  AddressPlus                 Value * Value -> Value;
```

In this part the intermediate language is defined. The language is strongly typed, however you may define only one type, which corresponds to an untyped language. BEG generates the back end in a way, that the front end can only produce correctly typed expression trees.

The definition of the intermediate language starts by introducing an identifier for each type. Then each intermediate operator is described. Therefore the types of the operands and optionally of the result have to be specified. If no result type is specified the operator is assumed to be a top level operator. Top level operators do not produce a result and can occur only in the root of an expression tree. Vice versa only these operators may occur in the root. By specifying the types of the operands also the arity of the operator is defined.

An operator may be defined as commutative by using the '+' instead of the '*' character to separate the operators. Commutative operators have to have arity two.

An operator might have some attributes. Attribute values are supplied by the front end and are stored internally. They can be accessed in conditions and emit–parts. Attributes have implementation language types whose names have to be specified in the description. User defined types can be used, however the user is responsible that these types are correctly imported.

There is small but important difference specifying a top level operator with or without the `->`. An operator with the `->` will not clear BEGs memory. This explained in section 6.2 in detail.

5.4 Register Set Description

Register_Set_Part ::=**REGISTERS** { Register_Definition //',' } ';'
Register_Definition ::=Register_Ident
['(' { Register_Ident // ',' } ')']';'

Example:

```
REGISTERS
  R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,
  D0(R0,R1),D2(R2,R3),D4(R4,R5),D6(R6,R7),D8(R8,R9),
  D10(R10,R11), D12(R12,R13), D14(R14,R15),
  F0, F1, F2, F3, F4, F5, F6, F7,
  DF0 (F0,F1), DF2(F2,F3), DF4(F4,F5), DF6(F6,F7);
```

The registers of the target machine are enumerated. A register might physically contain some other registers. This is specified enumerating the parts of an register in brackets. A register identifier has to occur exactly once in front of a bracket. Afterwards it may be contained several times in brackets. Only the general register allocator supports compound registers.

5.5 Nonterminal Definitions

Nonterminals_Part ::=**NONTERMINALS** {Nonterminal_Definition ';' }.
Nonterminal_Definition ::=Nonterminal_Ident
[**ADRMODE**
| **REGISTERS** '(' {Register_Ident //',' } ')']
[**COND_ATTRIBUTES** Attributes_Definitions ';']
[Attributes_Definitions].

Example:

```
Register   REGISTERS (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12);
RSAddress  ADRMODE   (a : GcgBase.Amode);
```

Each nonterminal has to be defined in this section. The register allocator distinguishes between three kinds of nonterminals: Addressingmode–, register– and memory–nonterminals. An addressingmode nonterminal is defined by the key word ADRMODE and a register nonterminal by the key word REGISTERS. In the latter case a default register set has to be given. If neither ADRMODE nor REGISTERS is specified the nonterminal is considered a memory nonterminal.

A nonterminal can have two kinds of attributes. The attributes defined immediately after COND_ATTRIBUTES are condition attributes and calculated during the first pass. Condition attributes have to be calculated by the EVAL part of the rules and might be used in the CONDITION part. The other type of attributes should be used whenever possible. These attributes are calculated during the output phase. They have to be calculated by the EMIT part of the rules and may not be used in the CONDITION part.

5.6 Rule Part

Rule_Part ::={ Rule | Routine}.
Rule ::=**RULE** Pattern ['->' Pattern_Nonterminal] ';'
[**CONDITION** Target_Text';']
COST Integer ';'
[**CHANGE** '(' { Register_Ident // ',' } ')' ';']
[**TARGET** Shortname ';']
[**EVAL** Target_Text';']
[**EMIT** Target_Text ';'].
Pattern ::=Operator_Ident ['.' Shortname] {Pattern} †
| Pattern_Nonterminal.
Pattern_Nonterminal ::=Nonterminal_Ident ['.' Shortname]
['(' { Register_Ident //',' } ')'].
Shortname ::=Ident.
Routine ::=**ROUTINE** Operator_Ident Target_Text ';'.

Example:

```
RULE   AddressPlus  Register (R1..R15)  Constant  -> RSAddress;
   CONDITION {(Constant.v>=0) AND (Constant.v<=4095)};
   COST 0;
   EMIT {WITH RSAddress.a DO base:=Register;
             offset:=Constant.v; END}
```

The number of patterns contained in †has to be equal to the arity of the operator. Every nonterminal and every operator contained in the rule gets a shortname. The scope of the shortname is the current rule. Shortnames have to be unique only if they are really used. If the shortname after an Operator_Ident or a Nonterminal_Ident is omitted it defaults to the Operator- or Nonterminal_Ident.

The Target_Text may contain references to attributes of operators and nonterminals. These references are written in the form ⟨Shortname⟩.⟨Attribute_Ident⟩. In the CONDITION or EVAL part only attributes of operators and condition attributes of nonterminals may be accessed.

Register nonterminals have an additional attribute register. It is computed by the register allocator. As it is a normal attribute of a nonterminal it may only be accessed in the EMIT part. After a register nonterminal in the pattern or as result nonterminal the admissible registers can be specified. The set defaults to the register set specified at the definition of the nonterminal. It is guaranteed that the value of the register attribute denotes one of these registers. The register attribute has the type Register which is defined in the definition module GcgBase. It is an enumeration type. Registers are named as described in the register set part proceeded by 'Reg'.

After the key word CHANGE the registers changed as a side effect of the current instruction may specified. The registers specified here have to be disjoint from the admissible registers for the operands or for the result.

Two address instructions can be described by TARGET. The shortname following the key word target denotes the operand whose register is also the result register. When using the on the fly register allocator the sets of admissible registers of the target operand and of the result have to be equal. For the general register allocator this assumption is relaxed. For each register r in one of the sets there has to exist a register s in the other set which is not disjoint from r. That means there is a register which is part of r and of s. The register allocator guarantees that the register picked for the target operand and the register picked for the result are not disjoint.

If the pattern contains commutative operators the rule is automatically duplicated with swapped operands. However it might happen that the pattern is symmetric. That means swapping of operands would just produce the same pattern again. In this case the rule is not duplicated. Usually this works fine but it might happen that the pattern is symmetric but the condition is not. In this case the rule has to be duplicated by the user.

Rules beginning with ROUTINE can be used to directly specify the target code which implements an intermediate operator. BEG just generates the procedure heading and places the target text inside. There may be at most one such rule per operator and operators occurring in those rules must not occur in any other pattern. This concept allows to transform the IR a bit by some hand written routines for example to split operators which are to complex. The target text can access the sons of the operator by writing op1, op2 ...and the attributes by prefixing the attribute name with 'At'.

5.7 Insertions Part

Insertions_Part ::=**INSERTS**
Insert_Ident { Target_Text }.
Insert_Ident ::=Ident.

This concept allows the user to insert arbitrary Target_Text into the modules generated by BEG. This allows to customize the code produced. There are certain insertion points contained in the code produced by BEG. They are documented in the next section. The Target_Text is inserted at the insertion point into the code produced by BEG.

5.8 Options

Options for BEG can be supplied in the command line or in the CGD. Options in the command line have precedence. They are inserted at the beginning of the CGD and are written by a '%' sign followed by the name of the option. If the option should be switched off the name has to be proceeded by 'no'. See section 6.5 for a description of the options available.

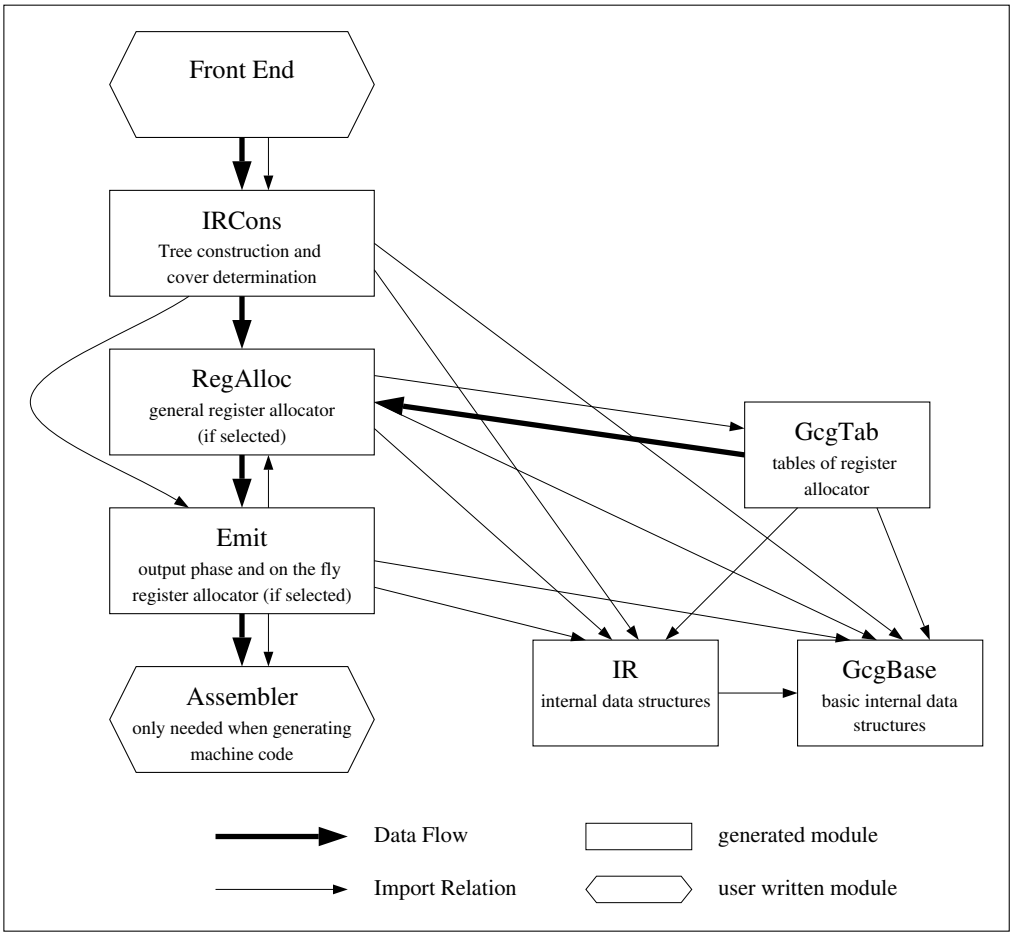

Figure 13: The structure of the GCG

6 The GCG

6.1 Structure of the GCG

Figure 13 shows the structure of the GCG in detail. It consists of the main interface module IRCons. The front end only knows this module. The other modules are hidden and theoretically the user need not know anything about these modules. However as BEG takes some implementation code out of the CGD and puts it into the module Emit, the user has to make sure that this module contains the necessary IMPORT statements. The modules GcgBase and IR contain definitions of internal data structures, which are of no interest for the normal user. The on the fly register allocator is directly generated into the Emit module, while the general register allocator is implemented by the modules RegAlloc and GcgTab.

The definition module IRCons does not contain any IMPORT statements. It changes only if the intermediate code part of the CGD is changed. So working on the rule part of the CGD does not require a recompilation of the front end. Actually the name of the

module IRCons is not fixed to 'IRCons'. Instead it gets the name of the CGD. This allows an easier interfacing of the front end. However we will use the name IRCons throughout this manual.

The GCG also uses the run time system modules Storage, System and InOut. InOut is only used if the test option is selected. Storage can be reassigned to a user written storage handler.

6.2 Interface to the Front End

The interface of the GCG is build by the module IRCons. First it defines each intermediate type a an opaque type. For each operator of the intermediate language there is one procedure in the interface :

PROCEDURE ⟨Operator⟩ (At⟨Attribute$_1$⟩: ⟨Target_Type$_1$⟩;
...
At⟨Attribute$_k$⟩: ⟨Target_Type$_l$⟩;
op1 : ⟨Intermediate_Type$_1$⟩;
...
opn : ⟨Intermediate_Type$_n$⟩;
VAR result : ⟨Intermediate_Type⟩);

n is the arity of the operator and k the number of its attributes. The result parameter is not present for operators without result.

The front end passes an expression tree in the following way. It starts with the leaves and calls the corresponding procedure. This returns a result. The result can be used as an operand for another operator by passing it as a parameter to the corresponding procedure. So the operators does not have to be passed strictly in postfix order, however a operator can be passed to the back end only after its operands have been passed too.

It is theoretically possible to use one subtree more than once. The current version of BEG does support this, however future versions might not. It also makes not much sense, because BEG generates the code twice rather than computing the value once and using it twice.

The front end has to pass one expression tree after the other. Calling of a procedure for a top level operator emits code for the current expression and afterwards clears the whole memory. This enables BEG to use a very fast allocation technique. However this restriction it sometimes very inconvenient for the front end. So it is relaxed a bit. A top level operator can be defined not to clear BEGs memory. Note that BEG can reuse the memory only if it has been cleared, so extensive use of the feature will lead to memory problems. However it is useful for small problems like extracting function calls out of expression trees. A top level operator is defined not to clear BEGs memory by writing the `->` in the operator definition.

There is an insertion point called *IpIRCons* which allows to extend the interface of the code generator.

6.3 The Import Interface

There are several possibilities to use external Modula identifiers in a CGD. These are the types of attributes and the external types, variables, constants and procedures used in the

Target_Text parts.

That is no problem at all, however the user has to take care of the necessary IMPORT statements. Therefore the following insertion points are provided:

- *IpTypes:* Types used as Attributes for Operators.
- *IpNtTypes:* Types used additionally as Attributes for Nonterminals. These may not include the types imported in IpTypes.
- *IpText:* Everything used in CONDITION, EVAL or EMIT parts.

6.4 Insertion Points

There is an insertion point in each module, directly behind the IMPORT statements. So the code inserted here might contain additional IMPORT statements and afterwards some arbitrary declarations. The name of these insertion points is Modulname_d or Modulname_i. The d means definition and the i means implementation module. There are some other insertion points which can be used to further customize the GCG:

- *IpIRCons:* goes into the definition module IRCons. It allows to extend the interface of the code generator by hand written procedures. The bodies of these procedures have to be inserted into IpIRCons_i.
- *IpEmit:* Definitions (constants, types, variables and procedures) needed by the emit parts.
- *IpInOut:* Defaults to 'FROM InOut IMPORT Write, WriteInt, WriteString, WriteCard, WriteLn' if the test option is selected otherwise it is empty. It is generated into the modules generating test output. One of them is Emit. You need this insertion if the CGD uses the InOut procedures whether the test option is selected or not. The insertion can also be used to process the test output by a user written module.
- *IpIRConsInit:* goes into the initialization routine of module IRCons.
- *IpEmitInit:* goes into the initialization routine of module Emit.
- *IpGcgBaseInit:* goes into the initialization routine of module GcgBase.
- *IpGcgTabInit:* goes into the initialization routine of module GcgTab.
- *IpIRConsStorage:* selects storage allocation module. It defaults to 'FROM Storage IMPORT ALLOCATE'.
- *IpEmitI1:* actions to perform after the build phase and before register allocation.
- *IpEmitI2:* actions to perform after register allocation and before the output phase.
- *IpNoCode:* action to perform if no code was selected because of an incomplete CGD. It defaults to a forced error (division by zero). If the test option is enabled test output is produced before the code of the insertion is executed.

- *IpNoReg:* action to perform if the register allocator blocks. If the test option is enabled a corresponding message is printed. *IpNoReg* defaults to a forced error by zero division.

- *IpGcgTypes:* User defined types can be used as attributes of operators or nonterminals. These types can be defined in a user module, in this insertion point, or in *IpIRCons* . Types defined in a user module have to be imported as described above. Types defined in *IpGcgTypes* have to be qualified with GcgBase and can only be used for attributes of nonterminals. Types defined in *IpIRCons_d* have to be qualified with IRCons (it might be necessary to import IRCons in *IpEmit_i* in that case).

6.5 Options

BEG options allow to control how the GCG is generated. They can be supplied in the command line or at the beginning of the CGD. The following options are available:

test / notest: Default is notest. If test is switched on routines for test output are generated. See section 6.6

Cro / noCro: Default is Cro. To switch on and off chain rule optimization. This option is of no interest to the normal user.

onthefly / noonthefly: Default is noonthefly. Selects the on the fly register allocator.

RegNameTable / noRegNameTable: Default is RegNameTable. A table 'RegNameTable : ARRAY Register OF ARRAY [0..9] OF CHAR' is generated. It is initialized with the register names specified in the CGD. It is located in GcgBase.md. This table is helpful when generating assembler code.

IRConsCheck / noIRConsCheck: Default is IRConsCheck. This option allows to switch off the type checking for the front end. Normally operands have opaque types. When Checking is off the all have the type ADDRESS.

6.6 The Test Output Interface

If the test option is enabled BEG will generate several routines to produce test output. Therefore BEG expects hand written print routines for each type used for attributes. These procedures have to be named 'Print' followed by the name of the type.

PROCEDURE PrintINTEGER (i : INTEGER);

BEG already supplies those routines for INTEGER, CARDINAL and BOOLEAN. The insertion *IpTestImport* has to contain IMPORT statements for the user written routines.

If the test option is enabled BEG will generate three boolean variables into the definition module GcgBase: OptEmitIR, OptEmitMatch and OptRegAlloc. The variables are initialized with FALSE. The user might assign TRUE if he wants to select a certain test output.

OptEmitIR After building the tree and before the register allocation and the output phase the intermediate tree is printed. It is printed in prefix notation. Each node uses one line for the operator name and the attributes. This option was used to produce the test output of the examples in A.1.4. Example

```
142954 Assign
143016   AddressPlus
143140     BlockBase
143078     Constant   4
143202   Constant   4011
```

The numbers denote internal tree pointers. They can be used to identify nodes when looking at the other test output.

OptEmitMatch This test output is a trace of the output phase. For each rule application four lines are printed. The first three lines are printed immediately before the emit action is executed the last line is printed afterwards.

Because the output phase executes the emit parts in postfix order this output also represents the tree in postfix order. However each block of four lines represents a rule application according to the minimal cover and not nodes of the intermediate tree.

The following information is printed: The first line contains the rule number and the line number where this rule starts in the CGD. Then the cost needed to cover the node and its descendants follows. Finally a summary of the rule is printed that is the root operator of the pattern and the result nonterminal.

The second line contains information about the tree node where the rule matches. The operator and its attributes are printed. Also the pointer is printed to get the relation to the other test output.

Then information about the register allocation is printed. The first number is of no interest for the user. Then the register assigned for the result is printed. If the result nonterminal is not a register nonterminal the register is undefined. After the slash the number of the spill location is printed. If it is not zero the register is spilled to that spill location after generation of the current instruction. Afterwards for each operand the assigned registers are printed. Note that registers are undefined (in this example NIL) if the corresponding nonterminal is not a register nonterminal.

Finally in the last line the attributes of the result nonterminal are printed. Because these are calculated by the emit part this line is printed after the emit part is executed. If the result nonterminal has no attributes an empty line is printed. In the examples in the appendix test output and code output mix up. So the generated code is printed before the fourth line of the test output is printed.

```
.  .  .  Rule 1/54 Cost=0 BlockBase -> Register
.  .  .  143140 BlockBase
.  .  .  ALLOC:  7 R13/0 Nil Nil
.  .  .
.  .  Rule 3/62 Cost=0 AddressPlus -> RSAddress
.  .  143016 AddressPlus
.  .  ALLOC:  6 Nil/0 R13 Nil
```

```
.   .  a=Amode 4(R13,R13)
.  Rule 6/81 Cost=0 RSAddress -> RXAddress
.  143016 AddressPlus
.  ALLOC:  5 Nil/0 Nil Nil
.  a=Amode 4(R13,R0)
.   .   .  Rule 7/85 Cost=0 Constant -> RSAddress
.   .   .  143202 Constant   4011
.   .   .  ALLOC:  4 Nil/0 Nil Nil
.   .   .  a=Amode 4011(R0,Nil)
.   .  Rule 6/81 Cost=0 RSAddress -> RXAddress
.   .  143202 Constant   4011
.   .  ALLOC:  3 Nil/0 Nil Nil
.   .  a=Amode 4011(R0,R0)
.  Rule 12/112 Cost=3 RXAddress -> Register
.  143202 Constant   4011
.  ALLOC:  2 R1/0 Nil Nil
.
Rule 21/154 Cost=7 Assign
Assign

ALLOC:  1 Nil/0 Nil R1
```

No Code Selected Test Output It might happen that it is impossible to translate a given IR statement with the rules of the CGD. In this case the GCG prints the message 'No code selected' and if test is enabled the following test output is produced:

```
 140818 Assign
 141126   AddressPlus
 141214     BlockBase
 141170     Constant   4
                    Register Cost=3 Rule=3
 140862   Plus
 140950     Content
                    Register Cost=4 Rule=6
 140994       AddressPlus
 141082         BlockBase
 141038         Constant   4
                      Register Cost=3 Rule=3
 140906     Constant   4711
```

It looks quite like the output produced by EmitIR. For each node of the tree one line is printed. Additionally for each node the following information is given. If there is a (minimal) cover of the subtree below the current node which leads to the nonterminal N a line is printed containing N, the cost of the cover and the number of the rule the cover starts with.

When analyzing this output the nodes without additional lines are most interesting. There exists no cover for the subtree below. In this example the Plus node is such a node.

However that error is implied by the fact that one son of Plus has no cover too. That is the Constant 4711. If you look at the corresponding CGD (see A.1.1) you will notice that there is only a rule for Constants in the range 0 . . . 4095.

6.7 The Spill Code Interface

The user has to provide three procedures. The register allocator calls them to emit register copy, spill or reload instructions. The register allocator assumes that these procedures are inserted with the insertion mechanism into the module Emit. It is possible to insert the complete procedures or just IMPORT statements for them.

PROCEDURE LR (to, from : Register);

has to produce code to copy the register named by to into the register from.

TYPE Spilllocation : INTEGER;
PROCEDURE Spill (reg : Register; loc : Spilllocation);
PROCEDURE Restore (reg : Register; loc : Spilllocation);

The procedures have to spill (reload) the contents of register reg into (from) the spill location. Spill and Restore are called in stack order so that they can be implemented directly by push and pop instructions. The parameter loc can be ignored in such cases. If the machine does not support a stack however the loc information can be used to simulate the stack in normal memory.

7 Installation and Usage

7.1 Introduction

This sections describes how to install and use BEG. It might not be up to date so please refer to the README file.

7.2 Files

With BEG you should receive the following files. Usually they are contained in a directory named beg with the subdirectories src, bin, sunmod, and example.

The source code, the object code of BEG (targeted for sun-3 machines), several examples, and some useful shell scripts are contained. The shell scripts should be seen as examples. It might be necessary to adapt them to your system.

README: Changes to this documentation.

src: Source code of BEG. It contains files with suffix .md definition modules and .mi implementation modules. Some module also have the suffix .dot. These files have to be processed with the preprocessor called dottool to get the corresponding .mi file. That does not matter to you if you just want to run BEG, because the result of the preprocessing is also included. However if you should want to make any changes it is very advisable to change the original sources rather than the output of the preprocessor.

src/Beg.mi: Main program of BEG.

src/dottool: Source code of the preprocessor. The preprocessor is described below in more detail. This directory also contains some shell scripts useful when using dottool with the Mocka compiler.

bin: Directory of executables.

bin/beg: A shell script which runs BEG. It has to be changed by the actual user.

bin/Beg: Binary file of BEG.

bin/dottool: Preprocessor needed by BEG.

sunmod: This subdirectory contains files you need to translate Beg with the sun Modula 2 compiler.

sunmod/makefile: Makefile used to translate BEG.

sunmod/SysDep.mod: This module has to be used for the sun Modula compiler.

sunmod/Exmake0?: Makefile to make the generated code generator. ? is 1,2,3, or 4 according to the sample CGD.

example: Contains several example CGDs and test drivers for them. The sample CGDs manex01.cgd ...manex04.cgd are the CGDs printed in this manual. Just run them through BEG and compile them. As main programs you can use Manex01.mi for manex01.cgd and Manex02.cgd for the other CGDs. The directory sunmod contains makefiles useful for that.

7.3 Usage of Beg

Beg expects two parameters: the name of the input file and the name of the output directory. It accepts the options described already also on the command line (with prefix '-'). However they do not have precedence over options specified in the source code.

If Beg runs successfully then the output directory contains all the modules of the GCG. It also contains a file named Stat.md with some statistical information.

If Beg fails it prints error messages consisting of line number, column number, and error text. The same information is also written into a file named ERRORS. It also calls a special procedure of the SysDep module in that case. The SysDep module for the Mocka compiler then uses some programs of the Mocka compiler to insert error messages into the source code and allows to edit it.

There is also a shell script named beg. It has the same parameters like Beg but works a bit different. It calls Beg and sends the output to a temporary directory. Then it compares each generated module to the one existing in the output directory. Only the really changed modules are copied. The advantage is that now make only recompiles the modules which really have to. The script also processes the generated module Emit with the preprocessor dottool (see below).

7.4 Adapting BEG to Other Modula Compilers

BEG contains one module called SysDep which contains all (currently known) dependencies of the generation (the machine BEG runs on) and the implementation machine (the machine the GCG runs on). The user who wants to port BEG on another machine (or just on another compiler) has to adopt this module. It is documented by comments in the source code.

Currently an incarnation of SysDep exists for the Mocka compiler and for the sun modula 2 compiler.

7.4.1 The Mocka Compiler

As BEG was developed with this compiler everything works out fine.

7.4.2 The Sun Modula Compiler

There is a directory for this compiler called beg/src/sunmod. It contains the body of the SysDep module and a makefile which allows to compile the Beg system. It also contains makefiles to compile the GCGs generated out of the example CGDs. As the number of generated modules change with the options of BEG the makefiles of the GCGs slightly differ.

7.5 Structure of the Source Code

Normally it is not necessary to do anything with the source code except to compile it. So this section is rather short. Just the main modules of BEG are described briefly:

IR: The data structure module of BEG. The structure of the internal representation of a CGD is implemented here.

Parser: Parses the CGD and builds the IR.

Semantic: Does semantic checks on the CGD and calculates informations needed for the generator modules.

GenX: Generates the module X of the GCG.

If it seem unavoidable to change the source code, please check if it is sufficient to only insert some more insertion points.

7.6 The Dot Tool

The dot tool is a small preprocessor which was used to build BEG. It allows to write programs which produce a lot of text output in a better syntax than Modula 2 does. The tool can also be used for the emit parts of rules when translating to assembler code. Then the module Emit which is generated by BEG has to be processed with the dottool afterwards.

Usage:

dottool < input > output

Dottool processes lines which have a dot in column one. All other lines are left unchanged. Lines beginning with '..' are control lines and are used to define parameters which control the processing of the other lines. Parameter names consist of one character. Upper and lower case letters are distinguished. Their value is a string of maximal 30 characters. The following line sets the parameter A to ' Test'.

..A Test

Lines beginning with a single dot are processed as follows. In the simplest case the line is translated to

WriteString('rest of the line'); WriteLn;

So if the line does not contain any specialities the generated program prints just the contents of the line (without the dot). If the line ends with a '-' character the final WriteLn is suppressed. The line might contain parts like '{c text}'. This construct is used to output values of variables instead of constant strings. If c=' ' text is copied unchanged into the generated program. Else c has to be the name of a parameter which has been set previously. The string value of this parameter is put into the generated program. If it contains a '%' sign this is replaced by text. For example if our program should print a line containing the value of an integer variable i we have two possibilities:

. beginning of line { WriteInt(i,1)} rest of line
or
..i WriteInt(%,1)
. beginning of line {ii} rest of line

The generated line looks like this (in both cases):

```
WriteString(" beginning of line "); WriteInt(i,1);
WriteString (" rest of line"); WriteLn;
```

The parameters % and $ have special meanings. They allow to redefine the strings 'WriteString' and 'WriteLn'. So other output procedures can be used. % defaults to 'WriteString(%)' and $ to 'WriteLn'. The characters '{', '\' and '-' at the end have to be escaped by an '\' if used as normal characters. The text contained in curly brackets may contain paired curly brackets. Unpaired brackets have to be escaped too.

The dottool keeps the line structure of the original source. For each input line exactly one output line is produced. So line numbers in error messages of the compiler stay correct.

References

[AGT87] *A.V. Aho, M. Ganapathi, S.W. Tjiang:* Code Generation Using Tree Matching and Dynamic Programming.

[ApSu87] *A.W. Appel, K.J. Supowit:* Generalizations of the Sethi–Ullman algorithm for register allocation. Software – Practice and Experience, Vol. 17(6), 417-421, June 1987

[Emme88] *H. Emmelmann:* Automatische Erzeugung effizienter Codegeneratoren. Diplomarbeit, GMD Studie 158, Gesellschaft fuer Mathematik und Datenverarbeitung mbH - Sankt Augustin, ISBN 3-88457-158-3

[ESL89] *H.Emmelmann, F-W.Schröer, R.Landwehr:* BEG – a Generator for Efficient Back Ends, Proceedings of the Sigplan'89 Conference on Programming Language Design and Implementation. Portland, Oregon, June 21–23, 1989, Sigplan Notices, Vol. 24, Number 7, July 1989

[GFH82] *M. Ganapathi, C.N. Fischer, J.L. Hennessy:* Retargetable Compiler Code Generation. Computing Surveys, Vol.14 No.4, Dec 82

[GaFi85] *M. Ganapathi, C.N. Fischer:* Affix Grammar Driven Code Generation. ACM Transactions on Programming Languages and Systems, Vol.7 No.4, Oct 85

[Glan78] *R.S. Glanville:* A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers. PhD Thesis, University of California, Berkeley, 1978

[GrHe84] *S.L. Graham, R.R. Henry et.al.:* Experience with a Graham–Glanville style code generator. Proceedings of the Sigplan 84 Symposium on Compiler Construction, Sigplan Notices, Vol. 19, Nr. 6

[Jans85] *H.-St. Jansohn:* Automated Generation of Optimized Code. GMD-Bericht Nr. 154, R.Oldenbourg Verlag, 1985

A Examples of Code Generator Descriptions

A.1 A Simple CGD

The aim of this example CGD is to show how BEG is used. It contains some of the rules used as examples in section 2. The CGD is very short so that it was possible to also include some modules produced by BEG. However this form of CGD is not a good example to demonstrate how to write good CGDs. It can not be extended straight forward to meet the requirements of a real compiler. The other CGD examples below illustrate how to design good CGDs.

A.1.1 CGD

```
(*   BEG Example CGD                                          *)
(*   Helmut Emmelmann 08/88                                   *)
(*   (c) GMD Forschungsstelle an der Universitaet Karlsruhe *)


(*   This example demonstrates the basic concepts of BEG.   *)
(*   The nonterminals and rules are selected only for that  *)
(*   purpose and not as an example for good CGD design.     *)
(*   The next example will show how a description look like *)
(*   which can be extended to describe the complete machine *)


%test    (* Option for BEG to generate test output routines *)
%RegNameTable

CODE_GENERATOR_DESCRIPTION  Example;
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant  ( v : INTEGER )                -> Value;
  Plus                        Value + Value -> Value;
  AddressPlus                 Value * Value -> Value;
  BlockBase                                 -> Value;
  Content                     Value         -> Value;
  Assign                      Value * Value;

REGISTERS
  R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15;

NONTERMINALS
  Register   REGISTERS (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12);

RULE   Plus
           Content
               AddressPlus
                  BlockBase
                  Constant
           Register.r            -> Register;
   COST 4; TARGET r;
   EMIT {WriteString ('   A    ');
         WrRegister (r.register); Write (',');
         WriteInt (Constant.v,1);
         WriteString ("(11)");
```

```
         WriteLn};

RULE   Constant                              -> Register;
   CONDITION {(Constant.v >=0) AND (Constant.v<=4095)}
   COST 3;
   EMIT {WriteString('   LA   '); WrRegister (Register.register);
         Write(","); WriteInt (Constant.v,1); WriteLn};

RULE   Assign
          AddressPlus
             BlockBase
             Constant
          Register.r;
   COST 4;
   EMIT {WriteString ('   ST    ');
         WrRegister (r.register); Write (',');
         WriteInt (Constant.v,1);
         WriteString ("(11)");
         WriteLn};


RULE   Plus    Register.s  Register.r      -> Register;
   COST 2;
   TARGET r;
   EMIT {WriteString ('   AR    ');
         WrRegister (r.register); Write (',');
         WrRegister (s.register); WriteLn};

RULE   Content
          AddressPlus
             BlockBase
             Constant                       -> Register.r;
   COST 4;
   EMIT {WriteString ('   L     ');
         WrRegister (r.register); Write (',');
         WriteInt (Constant.v,1); WriteString ("(11)");
         WriteLn};

INSERTS

IpInOut  {FROM    InOut IMPORT Write, WriteLn, WriteInt, WriteCard, WriteString;}

IpEmit_i {
    PROCEDURE WrRegister (r : Register);
    BEGIN
       WriteString (GcgBase.RegNameTable[r]);
       (* GcgBase.PrintRegister is the test output routine   *)
       (* for Register names generated by BEG                *)
    END WrRegister;

   PROCEDURE   LR  (to, from : Register);
   (*  Copy Register from into Register to *)
   BEGIN
      WriteString ('   LR    '); WrRegister(to); Write (',');
      WrRegister (from); WriteLn;
   END LR;
```

```
   PROCEDURE   Spill (reg : Register; loc : Spilllocation);
   BEGIN
      WriteString ('   Spill   '); WriteInt (loc,1); Write (',');
      WrRegister (reg); WriteLn;
   END Spill;

   PROCEDURE   Restore (reg : Register; loc : Spilllocation);
   BEGIN
      WriteString ('   Reload   '); WriteInt (loc,1); Write (',');
      WrRegister (reg); WriteLn;
   END Restore;
}
END CODE_GENERATOR_DESCRIPTION Example.
```

A.1.2 Module IRCons

This Module is the interface between the code generator and the front end.

```
 (*************************************************************)
 (*   This module was generated by BEG V1.0                   *)
 (*        GMD Forschunsstelle an der Universitaet Karlsruhe  *)
 (*        Haid- und Neu-Strasse 7  7500 Karlsruhe Germany    *)
 (*************************************************************)
DEFINITION MODULE IRCons;

(******* empty insertion IpTypes *******)
(******* empty insertion IpIRCons_d *******)

TYPE
          Value ;


(******* empty insertion IpIRCons *******)
PROCEDURE  Constant (
                    Atv  :  INTEGER
                    ;  VAR result : Value);
PROCEDURE  Plus (
                       op1  : Value
                    ;  op2  : Value
                    ;  VAR result : Value);
PROCEDURE  AddressPlus (
                       op1  : Value
                    ;  op2  : Value
                    ;  VAR result : Value);
PROCEDURE  BlockBase (
                       VAR result : Value);
PROCEDURE  Content (
                       op1  : Value
                    ;  VAR result : Value);
PROCEDURE  Assign (
                       op1  : Value
                    ;  op2  : Value
                                                    );


END IRCons.
```

A.1.3 Test Driver

```
(*******************************)
(* Test driver for Manex01.cgd  *)
(*******************************)

MODULE Manex01;
IMPORT  Example;
FROM    Example IMPORT Constant, Plus,
        Assign, AddressPlus, BlockBase,
        Content, Value;
IMPORT  GcgBase;

VAR   a,b,c,d,e,f,g,h,i,j,k,l,m,n,o
      : Value;
BEGIN
   GcgBase.OptEmitIR := TRUE;
(*   GcgBase.OptEmitMatch := TRUE;*)
   (*  Options for test output *)

   (*  (BB+4) := 4011 *)
      Constant (4011,c);
         BlockBase (e);
         Constant (4,d);
      AddressPlus (e,d,f);
   Assign (f,c);


(*  (BB+4) := Plus
      ( 1 Content
         (AddressPlus (BlockBase 4))) *)
         BlockBase (a);
         Constant  (4,b);
      AddressPlus (a,b,c);
               BlockBase (d);
               Constant  (4,e);
            AddressPlus (d,e,f);
         Content   (f,f);
         Constant (1,g);
      Plus (g,f,h);
   Assign (c,h);

(*  (BB+4) := Plus ( 1
   Plus (
      Content(AddressPlus(BlockBase 4))
      Content(AddressPlus(BlockBase 4))
        ))   *)
         BlockBase (a);
         Constant  (4,b);
      AddressPlus (a,b,c);
                  BlockBase (d);
                  Constant  (4,e);
               AddressPlus (d,e,f);
            Content (f,g);
                  BlockBase (h);
                  Constant  (4,i);
               AddressPlus (h,i,j);
            Content (j,l);
         Plus (g,l,m);
         Constant (1,n);
      Plus (n,m,o);
   Assign (c,o);

(*  There exists no cover for the
    following expression tree because
    there is no rule in Manex01.cgd
    for constants bigger than 4095.
    This demonstrates the test output
    of the GCG generated with the test
    option.

  (BB+4) := Plus  (Content
         (AddressPlus (BlockBase 4))
          4711)          *)
        BlockBase (a);
        Constant  (4,b);
     AddressPlus (a,b,c);
              BlockBase (d);
              Constant  (4,e);
           AddressPlus (d,e,f);
        Content (f,f);
        Constant (4711,g);
     Plus (f,g,h);
   Assign (c,h);

END Manex01.
```

A.1.4 Normal Test Output

```
141174 Assign
141218   AddressPlus
141306     BlockBase
141262     Constant   4
141350   Constant   4011
  LA  R0,4011
  ST   R0,4(11)
140954 Assign
141262   AddressPlus
141350     BlockBase
141306     Constant   4
140998   Plus
141042     Constant   1
141086     Content
141130       AddressPlus
141218         BlockBase
141174         Constant   4
  LA  R0,1
  A    R0,4(11)
  ST   R0,4(11)
140734 Assign
141262   AddressPlus
141350     BlockBase
141306     Constant   4
140778   Plus
140822     Constant   1
140866     Plus
141086       Content
141130         AddressPlus
141218           BlockBase
141174           Constant   4
140910       Content
140954         AddressPlus
141042           BlockBase
140998           Constant   4
  LA  R1,1
  L    R0,4(11)
  A    R0,4(11)
  AR   R0,R1
  ST   R0,4(11)
140954 Assign
141262   AddressPlus
141350     BlockBase
141306     Constant   4
```

```
 140998   Plus
 141086     Content
 141130       AddressPlus
 141218         BlockBase
 141174         Constant   4
 141042     Constant   4711
no code selected
 140954 Assign
 141262   AddressPlus
 141350     BlockBase
 141306     Constant   4
                    Register Cost=3 Rule=3
 140998   Plus
 141086     Content
                    Register Cost=4 Rule=6
 141130       AddressPlus
 141218         BlockBase
 141174         Constant   4
                    Register Cost=3 Rule=3
 141042     Constant   4711
```

A.1.5 Cover Test Output

```
 141038 Assign
 141082   AddressPlus
 141170     BlockBase
 141126     Constant   4
 141214   Constant   4011
.  Rule 3/46 Cost=3 Constant -> Register
.  141214 Constant   4011
.  ALLOC:  2 R0/0 Nil Nil
   LA  R0,4011
.
Rule 4/52 Cost=7 Assign
Assign

ALLOC:  1 Nil/0 R0 Nil

   ST   R0,4(11)
 140818 Assign
 141126   AddressPlus
 141214     BlockBase
 141170     Constant   4
 140862   Plus
 140906     Constant   1
 140950     Content
 140994       AddressPlus
 141082         BlockBase
 141038         Constant   4
.  .  Rule 3/46 Cost=3 Constant -> Register
.  .  140906 Constant   1
.  .  ALLOC:  3 R0/0 Nil Nil
   LA  R0,1
.  .
.  Rule 1/33 Cost=7 Plus -> Register
.  140862 Plus
.  ALLOC:  2 R0/0 R0 Nil
   A    R0,4(11)
.
Rule 4/52 Cost=11 Assign
Assign

ALLOC:  1 Nil/0 R0 Nil

   ST   R0,4(11)
 140598 Assign
 141126   AddressPlus
 141214     BlockBase
 141170     Constant   4
 140642   Plus
 140686     Constant   1
 140730     Plus
 140950       Content
 140994         AddressPlus
 141082           BlockBase
 141038           Constant   4
 140774       Content
 140818         AddressPlus
 140906           BlockBase
 140862           Constant   4
.  .  Rule 3/46 Cost=3 Constant -> Register
.  .  140686 Constant   1
.  .  ALLOC:  5 R1/0 Nil Nil
   LA  R1,1
.  .
.  .  .  Rule 6/72 Cost=4 Content -> Register
.  .  .  140950 Content
.  .  .  ALLOC:  4 R0/0 Nil Nil
   L    R0,4(11)
.  .  .
.  .  Rule 1/33 Cost=8 Plus -> Register
.  .  140730 Plus
.  .  ALLOC:  3 R0/0 R0 Nil
   A    R0,4(11)
.  .
.  Rule 5/65 Cost=13 Plus -> Register
.  140642 Plus
.  ALLOC:  2 R0/0 R1 R0
   AR   R0,R1
.
Rule 4/52 Cost=17 Assign
Assign

ALLOC:  1 Nil/0 R0 Nil

   ST   R0,4(11)
 140818 Assign
 141126   AddressPlus
 141214     BlockBase
 141170     Constant   4
 140862   Plus
 140950     Content
 140994       AddressPlus
 141082         BlockBase
 141038         Constant   4
 140906     Constant   4711
no code selected
 140818 Assign
 141126   AddressPlus
 141214     BlockBase
 141170     Constant   4
                    Register Cost=3 Rule=3
 140862   Plus
 140950     Content
                    Register Cost=4 Rule=6
 140994       AddressPlus
 141082         BlockBase
 141038         Constant   4
                    Register Cost=3 Rule=3
 140906     Constant   4711
```

A.2 IBM370 CGD without folding

The following example CGD can be extended straight forward to a complete IBM370 CGD. It handles both RX and RS Addresses as well as the double register problem. However it does not handle folding in addressing modes and not all types of big immediate operands. Folding in addressing modes is the following transformation. If a sum of addresses and offsets has to be calculated all constant values can be added at compile time and only the result added at run time. So the CGD assumes that a sum of addresses only contains one constant and this as the right operand of the last addressplus operator in this sum. This can be achieved by an easy transformation and is usually done anyway. This folding can also be done in a CGD, the next example demonstrates this. However then the CGD becomes a bit trickier.

A.2.1 CGD

```
(*   BEG Example CGD                                          *)
(*   Helmut Emmelmann 08/88                                   *)
(*   (c) GMD Forschungsstelle an der Universitaet Karlsruhe *)


(*   This is a more realistic CGD for IBM 370.                *)

(*   To keep the description simple the following             *)
(*   assumption is made:                                      *)
(*     constant folding in addressing modes  has already      *)
(*     been done. So if we have a sequence of AddressPlus     *)
(*     operators occurs in the input only the right son of    *)
(*     the last AddressPlus operator is a constant.           *)
(*   However this CGD only produces bad but not wrong code    *)
(*   if the assumption does not hold.                         *)
(*   The example in the next section describes how this       *)
(*   constant folding can be described within a CGD.          *)


%test    (* Option for BEG to generate test output routines *)


CODE_GENERATOR_DESCRIPTION  Example;
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant  ( v : INTEGER )                -> Value;
  Plus                        Value + Value -> Value;
  Mult                        Value + Value -> Value;
  Div                         Value * Value -> Value;
  AddressPlus                 Value * Value -> Value;
  BlockBase                                 -> Value;
  Content                     Value         -> Value;
  Assign                      Value * Value;

REGISTERS
  R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,
  D0(R0,R1),D2(R2,R3),D4(R4,R5),D6(R6,R7),D8(R8,R9),
  D10(R10,R11), D12(R12,R13), D14(R14,R15),
  F0, F1, F2, F3, F4, F5, F6, F7,
```

```
  DF0 (F0,F1), DF2(F2,F3), DF4(F4,F5), DF6(F6,F7);



NONTERMINALS
  Register   REGISTERS (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12);
  Double     REGISTERS (D0,D2,D4,D6,D8,D10);
  RSAddress  ADRMODE   (a : GcgBase.Amode);
  RXAddress  ADRMODE   (a : GcgBase.Amode);
  RegSum     ADRMODE   (r : Register; s : Register);
                (* Represents sum of r and s *)

(*  Addressing Modes *)


RULE   Register (R1..R15)                  -> RSAddress;
   COST 0;
   EMIT {WITH RSAddress.a DO base:=Register;
               offset:=0; END};

RULE   AddressPlus  Register (R1..R15)  Constant  -> RSAddress;
   CONDITION {(Constant.v>=0) AND (Constant.v<=4095)};
   COST 0;
   EMIT {WITH RSAddress.a DO base:=Register;
               offset:=Constant.v; END}

RULE   AddressPlus  BlockBase  Constant  -> RSAddress;
   CONDITION {(Constant.v>=0) AND (Constant.v<=4095)};
   COST 0;
   EMIT {WITH RSAddress.a DO base:=RegR13;
               offset:=Constant.v; END}

RULE   AddressPlus  Register.i (R1..R15) Register.b (R1..R15)  -> RegSum;
   COST 0;
   EMIT {RegSum.r := i; RegSum.s :=b};

RULE   AddressPlus  BlockBase Register.b (R1..R15)  -> RegSum;
   COST 0;
   EMIT {RegSum.r := RegR13; RegSum.s := b};

RULE   AddressPlus
           RegSum
           Constant                -> RXAddress;
   CONDITION {(Constant.v>=0) AND (Constant.v<=4095)};
   COST 0;
   EMIT {WITH RXAddress.a DO index:=RegSum.r; base:=RegSum.s;
               offset:=Constant.v; END};

RULE   RegSum                      -> RXAddress;
   COST 0;
   EMIT {WITH RXAddress.a DO index:=RegSum.r; base:=RegSum.s;
               offset:=0; END;};

RULE   RSAddress -> RXAddress;
   COST 0;
   EMIT {RXAddress.a := RSAddress.a; RXAddress.a.index := RegR0};
```

```
RULE   Constant  -> RSAddress;
   CONDITION {(Constant.v>=0) AND (Constant.v<=4095)};
   COST 0;
   EMIT {WITH RSAddress.a DO base:=RegR0;
              offset:=Constant.v; END}


(*   Double Register Chain Rules *)

RULE Double -> Register(R1,R3,R5,R7,R9,R11);
   COST 0; TARGET Double;

RULE Register (R0,R2,R4,R6,R8,R10) -> Double.d (D0,D2,D4,D6,D8,D10);
   COST 2; TARGET Register;
   EMIT {.      SRDA {*d},32}



(************************************************************)
RULE   Constant                              -> Register.r;
   COST 5;
   EMIT {.      L   {*r},=A({i Constant.v})};

RULE   Content RXAddress.a                   -> Register.r;
   COST 4;
   EMIT {.      L   {*r},{X a.a}}

RULE   RXAddress.a                           -> Register.r (R1..R12);
   COST 3;
   EMIT {.      LA  {*r},{X a.a}}



(*   Fixed Point Operations                                    *)

RULE   Plus  Content RXAddress.a
             Register.r                      -> Register;
   COST 4; TARGET r;
   EMIT {.      A   {*r},{X a.a}};

RULE   Plus    Register.s  Register.r        -> Register;
   COST 2;
   TARGET r;
   EMIT {.      AR  {*r},{*s}}


RULE Mult Register.a(R1,R3,R5,R7,R9,R11) Content RXAddress.b
                           -> Double.d (D0,D2,D4,D6,D8,D10);
   COST 20; TARGET a;
   EMIT {.      M   {*d},{X b.a}}

RULE Mult Register.a(R1,R3,R5,R7,R9,R11) Register.b
                           -> Double.d (D0,D2,D4,D6,D8,D10);
   COST 20; TARGET a;
   EMIT {.      MR  {*d},{*b}}
```

```
RULE Div  Double.d  Content RXAddress.b  -> Register(R1,R3,R5,R7,R9,R11);
   COST 20; TARGET d;
   EMIT {.      D   {*d},{X b.a}}

RULE Div  Double.d  Register.r -> Register(R1,R3,R5,R7,R9,R11);
   COST 20; TARGET d;
   EMIT {.      DR  {*d},{*r}}



(*   Stamtements                                          *)
RULE   Assign  RXAddress.a Register.r;
   COST 4;
   EMIT {.      ST  {*r},{X a.a}}


RULE   Assign  RSAddress.d Content RSAddress.s;
   COST 6;
   EMIT {s.a.index := RegR0;
.       MVC {S d.a},{X s.a}
        }



INSERTS

(*************************************************************)
(*  This CGD uses a user defined Modula type named Amode
    The type can be defined in a user module or in the
    insertions IpIR_d and IpIRCons. The first has to be
    used for types of nonterminal attributes and the
    other for operator attributes.
    The the type and a procedure for test output have to
    be imported.                                          *)

IpGcgTypes {  (* This record represents RS and RX Adresses.
             In the case of RS Adresses index is not used. *)
            TYPE   Amode = RECORD
             index,base : Register; offset : INTEGER
            END; }

IpTestImport { FROM  Prints02 IMPORT PrintAmode;}
     (* Import of user written test output routine *)

(*************************************************************)
(*  Routines which are used by the emit actions to output
    code. The dottool is used for better readablility     *)

IpEmit   {
(* Control lines for the dottool: *)
..* GcgBase.PrintRegister(%)
..S WrRSAddress(%)
..X WrRXAddress(%)
..i WriteInt (%,1)
```

```
 PROCEDURE WrRSAddress (a : GcgBase.Amode);
 BEGIN
.{ia.offset}-
       IF a.base#RegR0 THEN
.(4,{*a.base})-
(*          This line is expanded by the dottool to  *)
(*          Write ('('); WrRegister (a.base); Write (')');*)
       END;
 END WrRSAddress;

 PROCEDURE WrRXAddress (a : GcgBase.Amode);
 BEGIN
.{ia.offset}-
       IF a.base#RegR0 THEN
.({*a.base}-
          IF a.index#RegR0 THEN
.,{*a.index}-
          END;
.)-
       END;
 END WrRXAddress;

(*************************************************************)
(*   Routines needed by the Register Allocator               *)

   PROCEDURE   LR  (to, from : Register);
   (*  Copy Register from into Register to *)
   BEGIN
.       LR  {*to},{*from}
   END LR;

   PROCEDURE   Spill (reg : Register; loc : Spilllocation);
   BEGIN
.       ST  {*reg},SPL{iloc}
   END Spill;

   PROCEDURE   Restore (reg : Register; loc : Spilllocation);
   BEGIN
.       L   {*reg},SPL{iloc}
   END Restore;
}
(*************************************************************)
END CODE_GENERATOR_DESCRIPTION Example.
```

A.2.2 Test Driver

Is like the one for the previous example, but contains several more expression trees. It is therefore not printed here.

A.2.3 Test Output

```
143366 Assign
143434   AddressPlus
143570     BlockBase
143502     Constant   4
143638   Constant   4011
       LA  R1,4011
       ST  R1,4(R13)
143026 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
143094   Plus
143162     Constant   1
143230     Content
143298       AddressPlus
143434         BlockBase
143366         Constant   4
       LA  R1,1
       A   R1,4(R13)
       ST  R1,4(R13)
142686 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
142754   Plus
142822     Constant   1
142890     Plus
143230       Content
143298         AddressPlus
143434           BlockBase
143366           Constant   4
142958       Content
143026         AddressPlus
143162           BlockBase
143094           Constant   4
       LA  R1,1
       L   R0,4(R13)
       A   R0,4(R13)
       AR  R0,R1
       ST  R0,4(R13)
143026 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
143094   Plus
143230     Content
143298       AddressPlus
143434         BlockBase
143366         Constant   4
143162     Constant   4711
       L   R0,=A(4711)
       A   R0,4(R13)
       ST  R0,4(R13)
142958 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
143026   Div
143298     Mult
143434       Constant   4
143366       Constant   5
143094     Mult
143230       Constant   2
143162       Constant   5
       LA  R5,4
       LA  R1,5
       MR  D4,R1
       LA  R3,2
       LA  R1,5
       MR  D2,R1
       DR  D4,R3
       ST  R5,4(R13)
143162 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
143230   Content
143298     AddressPlus
143434       BlockBase
143366       Constant   8
       MVC 4(4,R13),8(R13)
143162 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
143230   Content
143298     AddressPlus
143434       BlockBase
143366       Constant   4711
       L   R1,=A(4711)
       L   R0,0(R1,R13)
       ST  R0,4(R13)
142822 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
142890   Content
142958     AddressPlus
143094       AddressPlus
143434         BlockBase
143162         Content
143230           AddressPlus
143366             BlockBase
143298             Constant   4
143026       Constant   12
       L   R1,4(R13)
       L   R0,12(R1,R13)
       ST  R0,4(R13)
142482 Assign
143502   AddressPlus
143638     BlockBase
143570     Constant   4
142550   Content
142618     AddressPlus
142754       Content
142822         AddressPlus
142958           AddressPlus
143434             BlockBase
143026             Mult
143094               Constant   4
143162               Content
143230                 AddressPlus
143366                   BlockBase
```

```
 143298                  Constant   4
 142890            Constant   12
 142686       Constant   64
        LA  R1,4
        M   D0,4(R13)
        L   R1,12(R1,R13)
        MVC 4(4,R13),64(R1)
```

A.2.4 Cover Test Output

This test output is not completely printed (because of its size).

```
 143366 Assign
 143434   AddressPlus
 143570     BlockBase
 143502     Constant   4
 143638   Constant   4011
. . Rule 3/67 Cost=0 AddressPlus -> RSAddress
. . 143434 AddressPlus
. . ALLOC:  6 Nil/0 Nil Nil
. . a=Amode 4(R13,R13)
. Rule 8/94 Cost=0 RSAddress -> RXAddress
. 143434 AddressPlus
. ALLOC:  5 Nil/0 Nil Nil
. a=Amode 4(R13,R0)
. . . Rule 9/98 Cost=0 Constant -> RSAddress
. . . 143638 Constant   4011
. . . ALLOC:  4 Nil/0 Nil Nil
. . . a=Amode 4011(R0,Nil)
. . Rule 8/94 Cost=0 RSAddress -> RXAddress
. . 143638 Constant   4011
. . ALLOC:  3 Nil/0 Nil Nil
. . a=Amode 4011(R0,R0)
. Rule 14/125 Cost=3 RXAddress -> Register
. 143638 Constant   4011
. ALLOC:  2 R1/0 Nil Nil
        LA  R1,4011
.
Rule 23/167 Cost=7 Assign
Assign

ALLOC:  1 Nil/0 Nil R1

        ST  R1,4(R13)



143026 Assign
 143502   AddressPlus
 143638     BlockBase
 143570     Constant   4
 143094   Plus
 143162     Constant   1
 143230     Content
 143298       AddressPlus
 143434         BlockBase
 143366         Constant   4
. . Rule 3/67 Cost=0 AddressPlus -> RSAddress
. . 143502 AddressPlus
. . ALLOC:  9 Nil/0 Nil Nil
. . a=Amode 4(R13,R13)
. Rule 8/94 Cost=0 RSAddress -> RXAddress
. 143502 AddressPlus
. ALLOC:  8 Nil/0 Nil Nil
. a=Amode 4(R13,R0)
. . . . Rule 9/98 Cost=0 Constant -> RSAddress
. . . . 143162 Constant   1
. . . . ALLOC:  7 Nil/0 Nil Nil
. . . . a=Amode 1(R0,R13)
. . . Rule 8/94 Cost=0 RSAddress -> RXAddress
. . . 143162 Constant   1
. . . ALLOC:  6 Nil/0 Nil Nil
. . . a=Amode 1(R0,R0)
. . Rule 14/125 Cost=3 RXAddress -> Register
. . 143162 Constant   1
. . ALLOC:  5 R1/0 Nil Nil
        LA  R1,1
. .
. . . Rule 3/67 Cost=0 AddressPlus -> RSAddress
. . . 143298 AddressPlus
. . . ALLOC:  4 Nil/0 Nil Nil
. . . a=Amode 4(R13,R0)
. . Rule 8/94 Cost=0 RSAddress -> RXAddress
. . 143298 AddressPlus
. . ALLOC:  3 Nil/0 Nil Nil
. . a=Amode 4(R13,R0)
. Rule 15/133 Cost=7 Plus -> Register
. 143094 Plus
. ALLOC:  2 R1/0 R1 Nil
        A   R1,4(R13)
.
Rule 23/167 Cost=11 Assign
Assign

ALLOC:  1 Nil/0 Nil R1

        ST  R1,4(R13)
```

A.3 IBM370 CGD with folding

The following example CGD can be extended straight forward to a complete IBM370 CGD. It handles both RX and RS Addresses as well as the double register problem. It also does folding in addressing modes. Therefore the concept of condition attributes is used which is not so nice on a theoretically point of view but works fine in practice.

A.3.1 CGD

```
(*   BEG Example CGD                                          *)
(*   Helmut Emmelmann 08/88                                   *)
(*   (c) GMD Forschungsstelle an der Universitaet Karlsruhe *)


(*   This is a more realistic CGD for IBM 370.                *)
(*   It handles even folding in addressing modes well,        *)
(*   the double register problem and addressing beyond        *)
(*   the 4096 boundary.                                       *)



%test          (* Option for BEG to generate test output routines       *)
%RegNameTable  (* produce a table of register names for output routines *)


CODE_GENERATOR_DESCRIPTION  Example;
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant  ( v : INTEGER )                -> Value;
  Plus                         Value + Value -> Value;
  Mult                         Value + Value -> Value;
  Div                          Value * Value -> Value;
  AddressPlus                  Value + Value -> Value;
  BlockBase                                  -> Value;
  Content                      Value         -> Value;
  Assign                       Value * Value;

REGISTERS
  R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,
  D0(R0,R1),D2(R2,R3),D4(R4,R5),D6(R6,R7),D8(R8,R9),
  D10(R10,R11), D12(R12,R13), D14(R14,R15),
  F0, F1, F2, F3, F4, F5, F6, F7,
  DF0 (F0,F1), DF2(F2,F3), DF4(F4,F5), DF6(F6,F7);



NONTERMINALS
  Register   REGISTERS (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12);
  RegOfs     REGISTERS (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12)
             (offset : INTEGER);
  Double     REGISTERS (D0,D2,D4,D6,D8,D10);
  ContRS     ADRMODE   (a : GcgBase.Amode);
  ContRX     ADRMODE   (a : GcgBase.Amode);
  RSAddr     ADRMODE   (a : GcgBase.Amode);
  RXAddr     ADRMODE   (a : GcgBase.Amode);
  AbsRS      ADRMODE   COND_ATTRIBUTES (offset : INTEGER)
```

```
                    (base : Register);
  Const      COND_ATTRIBUTES  (v : INTEGER);
  AbsRX      ADRMODE   COND_ATTRIBUTES (offset : INTEGER)
                    (base : Register; index : Register);



(*  Folding *)
RULE   Constant                              -> Const;
   COST 0;
   EVAL {Const.v := Constant.v};

RULE   AddressPlus  Const.a Constant         -> Const;
   COST 0;
   EVAL {Const.v := a.v+Constant.v};


(*  Addressing Modes *)
RULE   BlockBase                             -> AbsRS;
   COST 0;       (* dedicated Register for Block Base *)
   EVAL {AbsRS.offset := 0};
   EMIT {AbsRS.base := RegR13};

RULE   Const                                 -> AbsRS;
   COST 0;
   EVAL {AbsRS.offset := Const.v};
   EMIT {AbsRS.base := RegR0};

RULE   Register (R1..R12)                    -> AbsRS;
   COST 0;
   EVAL {AbsRS.offset := 0};
   EMIT {AbsRS.base := Register};

RULE   AddressPlus  AbsRS  Register (R1..R12)  -> AbsRX;
   COST 1;
   EVAL {AbsRX.offset := AbsRS.offset};
   EMIT {AbsRX.base   := AbsRS.base;
         AbsRX.index  := Register};

RULE   AddressPlus  AbsRS.a  Const           -> AbsRS;
   COST 0;
   EVAL {AbsRS.offset := a.offset+Const.v};
   EMIT {AbsRS.base   := a.base};


RULE   AddressPlus  AbsRX.a  Const           -> AbsRX;
   COST 0;
   EVAL {AbsRX.offset := a.offset+Const.v};
   EMIT {AbsRX.base   := a.base; AbsRX.index := a.index};

RULE   AbsRS                                 -> AbsRX;
   COST 0;
   EVAL {AbsRX.offset := AbsRS.offset};
   EMIT {AbsRX.base   := AbsRS.base;
         AbsRX.index  := RegR0};
```

```
RULE   AbsRS                                -> RSAddr;
   CONDITION {(AbsRS.offset>=0) AND (AbsRS.offset<=4095)};
   COST 0;
   EMIT {WITH RSAddr.a DO offset := AbsRS.offset;
            base := AbsRS.base; index := RegR0 END};

RULE   AbsRX                                -> RXAddr;
   CONDITION {(AbsRX.offset>=0) AND (AbsRX.offset<=4095)};
   COST 0;
   EMIT {WITH RXAddr.a DO offset := AbsRX.offset;
            base := AbsRX.base; index := AbsRX.index END};

RULE   Content RSAddr -> ContRS;
   COST 0;
   EMIT {ContRS.a := RSAddr.a};

RULE   Content RXAddr -> ContRX;
   COST 0;
   EMIT {ContRX.a := RXAddr.a};

RULE   AbsRS                                -> RegOfs;
   COST 6;
   EMIT {
      IF RegOfs # AbsRS.base THEN
.       LR    {*RegOfs},{*AbsRS.base}
      END;
.       A     {*RegOfs},=A({i 4096*(AbsRS.offset DIV 4096)})
      RegOfs.offset := AbsRS.offset MOD 4096;
         };

RULE   RegOfs (R1..R12)                     -> RSAddr;
   COST 0;
   EMIT {WITH RSAddr.a DO base := RegOfs; offset := RegOfs.offset;
                          index := RegR0; END};

(* This rule does not work! *)
(* No code may be emitted within addressing mode rules *)
(* to overcome this problem it is possible to extend   *)
(* the attributes of RXAddr and to emit the instruction*)
(* in the rule which uses the RXAddr.                  *)
(*
RULE   AbsRS                                -> RXAddr;
   COST 4;
   EMIT {
.       L     R15,=A({i 4096*(AbsRS.offset DIV 4096)})
        WITH RXAddr.a DO
           index := RegR15; base := AbsRS.base;
           offset := AbsRS.offset MOD 4096;
        END };
*)


RULE   ContRS                               -> ContRX;
   COST 0;
   EMIT {ContRX.a := ContRS.a;}

RULE   RSAddr                               -> RXAddr;
```

```
   COST 0;
   EMIT {RXAddr.a := RSAddr.a};


(*   Double Register Chain Rules *)

RULE Double -> Register(R1,R3,R5,R7,R9,R11);
   COST 0; TARGET Double;

RULE Register (R0,R2,R4,R6,R8,R10) -> Double.d (D0,D2,D4,D6,D8,D10);
   COST 2; TARGET Register;
   EMIT {.      SRDA {*d},32}



(************************************************************)
RULE   Constant                            -> ContRS.r;
   COST 0;
   EMIT {WITH r.a DO base:=RegNil; offset:= Constant.v; END};

RULE   ContRX.a                            -> Register.r;
   COST 4;
   EMIT {.      L   {*r},{X a.a}}

RULE   RXAddr.a                            -> Register.r (R0..R12);
   COST 3;
   EMIT {.      LA  {*r},{X a.a}}



(*   Fixed Point Operations                                  *)

RULE   Plus  ContRX.a
             Register.r                    -> Register;
   COST 4; TARGET r;
   EMIT {.      A   {*r},{X a.a}};

RULE   Plus    Register.s  Register.r      -> Register;
   COST 2;
   TARGET r;
   EMIT {.      AR  {*r},{*s}}


RULE Mult Register.a(R1,R3,R5,R7,R9,R11) ContRX.b
                           -> Double.d (D0,D2,D4,D6,D8,D10);
   COST 20; TARGET a;
   EMIT {.      M   {*d},{X b.a}}

RULE Mult Register.a(R1,R3,R5,R7,R9,R11) Register.b
                           -> Double.d (D0,D2,D4,D6,D8,D10);
   COST 18; TARGET a;
   EMIT {.      MR  {*d},{*b}}


RULE Div  Double.d  ContRX.b  -> Register(R1,R3,R5,R7,R9,R11);
   COST 20; TARGET d;
   EMIT {.      D   {*d},{X b.a}}
```

```
RULE Div  Double.d  Register.r -> Register(R1,R3,R5,R7,R9,R11);
   COST 20; TARGET d;
   EMIT {.      DR  {*d},{*r}}



(*   Stamtements                                          *)
RULE   Assign  RXAddr.a Register.r;
   COST 4;
   EMIT {.      ST  {*r},{X a.a}}


RULE   Assign  RSAddr.d ContRS.s;
   COST 6;
   EMIT {s.a.index := RegR0;
.       MVC {S d.a},{X s.a}
        }



INSERTS

(*************************************************************)
(*  This CGD uses a user defined Modula type named Amode
    The type can be defined in a user module or in the
    insertions IpIR_d and IpIRCons. The first has to be
    used for types of nonterminal attributes and the
    other for operator attributes.
    The the type and a procedure for test output have to
    be imported.                                          *)

IpGcgTypes {  (* This record represents RS and RX Adresses.
             In the case of RS Adresses index is not used. *)
            TYPE   Amode = RECORD
             index,base : Register; offset : INTEGER
            END; }

IpTestImport { FROM  Prints02 IMPORT PrintAmode;}
     (* Import of user written test output routine *)

(*************************************************************)
(*  Routines which are used by the emit actions to output
    code. The dottool is used for better readablility     *)

IpEmit   {
(* Control lines for the dottool: *)
..* WrRegister(%)
..S WrRSAddress(%)
..X WrRXAddress(%)
..i WriteInt (%,1)

 PROCEDURE WrRegister (r : Register);
 BEGIN
    WriteString (GcgBase.RegNameTable[r]);
 END WrRegister;
```

```
 PROCEDURE WrRSAddress (a : GcgBase.Amode);
 BEGIN
.{ia.offset}-
       IF a.base#RegR0 THEN
.(4,{*a.base})-
(*          This line is expanded by the dottool to  *)
(*          Write ('('); WrRegister (a.base); Write (')');*)
       END;
 END WrRSAddress;

 PROCEDURE WrRXAddress (a : GcgBase.Amode);
 BEGIN
    IF a.base=RegNil THEN
       (* This indicated an immediate operand *)
.=A({ia.offset})-
    ELSE
.{ia.offset}-
       IF (a.base#RegR0) OR (a.index#RegR0) THEN
.({*a.base}-
          IF a.index#RegR0 THEN
.,{*a.index}-
          END;
.)-
       END;
    END;
 END WrRXAddress;

(*************************************************************)
(*   Routines needed by the Register Allocator               *)

   PROCEDURE   LR  (to, from : Register);
   (*  Copy Register from into Register to *)
   BEGIN
.       LR  {*to},{*from}
   END LR;

   PROCEDURE   Spill (reg : Register; loc : Spilllocation);
   BEGIN
.       ST  {*reg},SPL{iloc}
   END Spill;

   PROCEDURE   Restore (reg : Register; loc : Spilllocation);
   BEGIN
.       L   {*reg},SPL{iloc}
   END Restore;
}
(*************************************************************)

IpInOut {
    FROM    InOut IMPORT Write, WriteLn, WriteInt, WriteCard, WriteString;}

END CODE_GENERATOR_DESCRIPTION Example.
```

A.3.2 Test Output

```
145138 Assign
145248   AddressPlus
145468     BlockBase
145358     Constant   4
145578   Constant   4011
       MVC 4(4,R13),=A(4011)
144588 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
144698   Plus
144808     Constant   1
144918     Content
145028       AddressPlus
145248         BlockBase
145138         Constant   4
       LA  R0,1
       A   R0,4(R13)
       ST  R0,4(R13)
144038 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
144148   Plus
144258     Constant   1
144368     Plus
144918       Content
145028         AddressPlus
145248           BlockBase
145138           Constant   4
144478       Content
144588         AddressPlus
144808           BlockBase
144698           Constant   4
       L   R0,4(R13)
       A   R0,4(R13)
       A   R0,=A(1)
       ST  R0,4(R13)
144588 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
144698   Plus
144918     Content
145028       AddressPlus
145248         BlockBase
145138         Constant   4
144808     Constant   4711
       L   R0,4(R13)
       A   R0,=A(4711)
       ST  R0,4(R13)
144478 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
144588   Div
145028     Mult
145248       Constant   4
145138       Constant   5
144698     Mult
144918       Constant   2
144808       Constant   5
       LA  R3,5
       M   D2,=A(4)
       LA  R1,5
       M   D0,=A(2)
       DR  D2,R1
       ST  R3,4(R13)
144808 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
144918   Content
145028     AddressPlus
145248       BlockBase
145138       Constant   8
       MVC 4(4,R13),8(R13)
144808 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
144918   Content
145028     AddressPlus
145248       BlockBase
145138       Constant   4711
       LR    R1,R13
       A     R1,=A(4096)
       MVC 4(4,R13),615(R1)
144258 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
144368   Content
144478     AddressPlus
144698       AddressPlus
145248         BlockBase
144808         Content
144918           AddressPlus
145138             BlockBase
145028             Constant   4
144588       Constant   12
       L   R1,4(R13)
       L   R0,12(R13,R1)
       ST  R0,4(R13)
143708 Assign
145358   AddressPlus
145578     BlockBase
145468     Constant   4
143818   Content
143928     AddressPlus
144148       Content
144258         AddressPlus
144478           AddressPlus
145248             BlockBase
144588             Mult
144698               Constant   4
144808               Content
144918                 AddressPlus
145138                   BlockBase
145028                   Constant   4
144368           Constant   12
144038       Constant   64
       LA  R1,4
       M   D0,4(R13)
       L   R1,12(R13,R1)
       MVC 4(4,R13),64(R1)
```

A.4 MC68020 CGD

The following example CGD can be extended straight forward to a complete MC68020 CGD. In fact it was derived from our real MC68020 CGD for the Mocka compiler. It contains all the nonterminals needed to describe the whole processor (except Freg for floating point registers), however the IR is much simpler. Nevertheless the complete addressing mode capabilities are used (to the extend they are useful for the compiler): The following things have been left out:

- most of the IR operators.
- floating point operations.
- the CGD always uses mode long.
- special optimizations like inc, addq or shifting instead of multiplication.
- the emit parts are producing symbolic code rather than calling an assembler module which produces machine code.

However as the nonterminals were kept it is possible to extend the CGD straight forward to a complete CGD.

Note that this CGD uses condition attributes though it would not have to. The condition attributes are needed when folding is done and it has to be checked if the result is in a certain range. This check is necessary for example when the offset in an addressing mode has a limited size. However that is not true on the MC68020. For each addressing mode with a 16 or 8 bit displacement there is also one with a 32 bit displacement. This is handled by the assembler so no check is needed in the CGD. However a CGD for the MC68000 would have to check because the 32 bit displacement is not available. As we wanted to be able to derive a MC68000 CGD by small changes only the MC68020 CGD also uses condition attributes.

A.4.1 CGD

```
(*   BEG Example CGD                                          *)
(*   Helmut Emmelmann 08/88                                   *)
(*   (c) GMD Forschungsstelle an der Universitaet Karlsruhe *)


(*   This is a realistic CGD for MC68020                      *)



%onthefly (* On the fly register allocation possible         *)
%test     (* Option for BEG to generate test output routines *)
%RegNameTable


CODE_GENERATOR_DESCRIPTION  Example;
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant  ( v : INTEGER )                -> Value;
```

```
  Plus                        Value + Value -> Value;
  Mult                        Value + Value -> Value;
  Div                         Value * Value -> Value;
  AddressPlus                 Value + Value -> Value;
  BlockBase                                 -> Value;
  Content                     Value         -> Value;
  Assign                      Value * Value;


REGISTERS
(*******)
  d0,d1,d2,d3,d4,d5,d6,d7, a0,a1,a2,a3,a4,a5,a6,a7;

NONTERMINALS
(**********)

  Areg  REGISTERS (a0,a1,a2,a3,a4,a5,a6,a7);
  Dreg  REGISTERS (d0,d1,d2,d3,d4,d5,d6,d7);

  AregDispl ADRMODE COND_ATTRIBUTES (val  : LONGINT)
            (base  : Register);
  AregIndex ADRMODE COND_ATTRIBUTES (val  : LONGINT)
            (base  : Register;
             index : Register;
             scale : INTEGER);
  IndDispl  ADRMODE
            (val      : INTEGER;
             valinner : INTEGER;
             base     : Register);
  IndIndex  ADRMODE
            (val      : INTEGER;
             valinner : INTEGER;
             base     : Register;
             index    : Register;
             scale    : INTEGER;
             preindex : BOOLEAN);
  Dest      ADRMODE (op    : Operand);
  ea        ADRMODE (op    : Operand);
  Const     COND_ATTRIBUTES (val  : INTEGER);



(*  Chain RULEs ---------------------------------------------------------- *)
RULE   Areg                   ->   AregDispl;
  COST 0;
  EVAL  {AregDispl.val := 0};
  EMIT  {AregDispl.base := Areg.register;};

RULE   Areg                   ->   Dest;
  COST 0;
  EMIT {. {*Areg}@{= Dest.op}-};

RULE   Areg                   ->   ea;
  COST 0;
  EMIT {. {*Areg}{=ea.op}-};

RULE   Dreg                   ->   ea;
```

```
  COST 0;
  EMIT {. {*Dreg}{=ea.op}-};

RULE   AregDispl.a            ->   Dest;
  COST 2;
  EMIT {. {*a.base}@({i a.val}){= Dest.op}-};

RULE   AregIndex.a            ->   Dest;
  COST 2;
  EMIT {. {*a.base}@({i a.val},{* a.index}:l:{i a.scale}){= Dest.op}-};

RULE   IndDispl.a             ->   Dest;
   COST 4;
   EMIT {. {*a.base}@({i a.valinner})@({i a.val}){=Dest.op}-};

RULE   IndIndex.a             ->   Dest;
   COST 4;
   EMIT {IF a.preindex THEN
. {*a.base}@({i a.valinner},{*a.index}:l:{i a.scale})@({i a.val}){=Dest.op}-
     ELSE
. {*a.base}@({i a.valinner})@({i a.val},{*a.index}:l:{i a.scale}){=Dest.op}-
     END;};

RULE   Dest                   ->   Areg (a0..a5);
  COST 5;
  EMIT {.       lea     {s Dest.op},{*Areg}}

RULE   ea                     ->   Areg (a0..a5);
  COST 6;
  EMIT {.       movl    {s ea.op},{*Areg}}

RULE   ea                     ->   Dreg;
  COST 6;
  EMIT {.       movl    {s ea.op},{*Dreg}}


RULE   Const                  ->   ea;
  COST 0;
  EMIT {. #{i Const.val}{= ea.op}-};

(*  Address Calculations ------------------------------------------------ *)
RULE   AddressPlus    AregDispl.a   Const.o -> AregDispl.r;
  COST 0;
  EVAL  {r.val := a.val + o.val};
  EMIT  {r.base := a.base;};

RULE   AddressPlus    AregIndex.a   Const.o -> AregIndex.r;
  COST 0;
  EVAL  {r.val := a.val + o.val};
  EMIT  {r.scale := a.scale; r.index := a.index; r.base  := a.base;};


RULE   AddressPlus    IndDispl.a   Const.o -> IndDispl.r;
  COST 0;
  EMIT  {r.val := a.val + o.val; r.valinner := a.valinner;
         r.base  := a.base;};
```

```
RULE   AddressPlus    IndIndex.a   Const.o -> IndIndex.r;
  COST 0;
  EMIT  {r.val:= a.val+o.val; r.valinner:= a.valinner;
         r.scale := a.scale; r.index:=a.index;
         r.preindex := a.preindex; r.base:=a.base;}

RULE   AddressPlus    AregDispl.a   Dreg      -> AregIndex.r;
  COST 2;
  EVAL {r.val := a.val};
  EMIT {r.scale    := 1; r.index    := Dreg.register; r.base := a.base;}

RULE   AddressPlus    IndDispl.a   Dreg      -> IndIndex.r;
  COST 2;
  EMIT {r.val := a.val; r.valinner := a.valinner; r.base:=a.base;
        r.index    := Dreg; r.scale    := 1; r.preindex:=FALSE }

RULE   AddressPlus    AregDispl.a   Mult  Dreg  Const-> AregIndex.r;
   CONDITION {(Const.val=1) OR (Const.val=2) OR
              (Const.val=4) OR (Const.val=8)};
   COST 2;
   EVAL {r.val := a.val;};
   EMIT {r.scale := Const.val;
         r.index := Dreg.register; r.base := a.base;};

RULE   AddressPlus    IndDispl.a   Mult  Dreg  Const-> IndIndex.r;
   CONDITION {(Const.val=1) OR (Const.val=2) OR
              (Const.val=4) OR (Const.val=8)};
   COST 2;
   EMIT {r.val := a.val; r.valinner := a.valinner; r.base:=a.base;
         r.index := Dreg; r.scale := Const.val; r.preindex := FALSE};

RULE   Content AregDispl.a -> IndDispl.r;
   COST 0;
   EMIT {r.val  := 0; r.valinner := a.val; r.base     := a.base;};

RULE   Content AregIndex.a -> IndIndex.r;
   COST 0;
   EMIT { r.valinner := a.val; r.val      := 0; r.scale    := a.scale;
          r.index    := a.index; r.preindex := TRUE; r.base:=a.base};

RULE   Content  Dest     ->  ea;
   COST 2;
   EMIT {  ea.op := Dest.op; };

RULE   BlockBase -> AregDispl.r;
   COST 0;
   EVAL {r.val:=0};
   EMIT {r.base := Rega6};

(* other operators ----------------------------------------*)
RULE   Constant             ->  Const;
   COST 0;
   EVAL { Const.val := Constant.v};

RULE    Plus   Dreg.a  ea         -> Dreg;
   COST 6; TARGET a;
   EMIT {.      addl    {s ea.op},{*a}};
```

```
RULE    Mult   Const.a Const.b -> Const.r;
   COST 0;
   EVAL {r.val := a.val * b.val};

RULE    Mult   Dreg.a  ea         -> Dreg;
   COST 20; TARGET a;
   EMIT {.      muls    {s ea.op},{*a}};

RULE    Div    Dreg.a  ea         -> Dreg;
   COST 20; TARGET a;
   EMIT {.      divs    {s ea.op},{*a}};


(*  Statements ------------------------------------------------------------ *)
RULE    Assign      Dest    ea;
   COST 4;
   EMIT {.      movl    {s ea.op},{s Dest.op}};

RULE    Assign      Dest    Const;
   CONDITION {Const.val=0};
   COST 2;
   EMIT {.      clrl    {s Dest.op}};

INSERTS
IpEmit_i {FROM  GenOut  IMPORT  GenInt, GetLine, GenString, GenLn;}

IpEmit   {
(* Control lines for the dottool: *)
..* GenRegister(%)
..i GenInt (%)
..= GetLine(%)
..s GenString(%)
..$ GenLn
..% GenString('%')

TYPE      Operand = ARRAY [0..80] OF CHAR;
PROCEDURE PrintOperand (o : Operand);
BEGIN
   WriteString (o);
END PrintOperand;

PROCEDURE GenRegister (r : Register);
BEGIN
   GenString (GcgBase.RegNameTable[r]);
END GenRegister;
(*************************************************************)
(*   Routines needed by the Register Allocator               *)

   PROCEDURE   LR  (to, from : Register);
   (*  Copy Register from into Register to *)
   BEGIN
.       movl  {*from},{*to}
   END LR;

   PROCEDURE   Spill (reg : Register; loc : Spilllocation);
   BEGIN
```

```
.       pushl  {*reg}
   END Spill;

   PROCEDURE   Restore (reg : Register; loc : Spilllocation);
   BEGIN
.       popl   {*reg}
   END Restore;
}
(*************************************************************)

END CODE_GENERATOR_DESCRIPTION Example.
```

A.4.2 Test Output

```
137862 Assign
137966   AddressPlus
138174     BlockBase
138070     Constant   4
138278   Constant   4011
       movl     #4011, a6@(4)
137342 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
137446   Plus
137550     Constant   1
137654     Content
137758       AddressPlus
137966         BlockBase
137862         Constant   4
       movl     a6@(4),d0
       addl     #1,d0
       movl     d0, a6@(4)
136822 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
136926   Plus
137030     Constant   1
137134     Plus
137654       Content
137758         AddressPlus
137966           BlockBase
137862           Constant   4
137238       Content
137342         AddressPlus
137550           BlockBase
137446           Constant   4
       movl     a6@(4),d0
       addl     a6@(4),d0
       addl     #1,d0
       movl     d0, a6@(4)
137342 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
137446   Plus
137654     Content
137758       AddressPlus
137966         BlockBase
137862         Constant   4
137550     Constant   4711
       movl     #4711,d0
       addl     a6@(4),d0
       movl     d0, a6@(4)
137238 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
137342   Div
137758     Mult
137966       Constant   4
137862       Constant   5
137446     Mult
137654       Constant   2
137550       Constant   5
       movl     #20,d0
       divs     #10,d0
       movl     d0, a6@(4)
137550 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
137654   Content
137758     AddressPlus
137966       BlockBase
137862       Constant   8
       movl     a6@(8), a6@(4)
137550 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
137654   Content
137758     AddressPlus
137966       BlockBase
137862       Constant   4711
       movl     a6@(4711), a6@(4)
137030 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
137134   Content
137238     AddressPlus
137446       AddressPlus
137966         BlockBase
137550         Content
137654           AddressPlus
137862             BlockBase
137758             Constant   4
137342       Constant   12
       movl     a6@(4),d0
       movl     a6@(12,d0:l:1), a6@(4)
136510 Assign
138070   AddressPlus
138278     BlockBase
138174     Constant   4
136614   Content
136718     AddressPlus
136926       Content
137030         AddressPlus
137238           AddressPlus
137966             BlockBase
137342             Mult
137446               Constant   4
137550               Content
137654                 AddressPlus
137862                   BlockBase
137758                   Constant   4
137134           Constant   12
136822       Constant   64
       movl     a6@(4),d0
       movl     a6@(12,d0:l:4)@(64), a6@(4)
```