# BEG – a Generator for Efficient Back Ends

*Helmut Emmelmann, Friedrich–Wilhelm Schröer, Rudolf Landwehr*

GMD Forschungsstelle an der Universität Karlsruhe
Haid-und-Neu-Str. 7, D–7500 Karlsruhe 1, Germany

## Abstract

This paper describes a system that generates compiler back ends from a strictly declarative specification of the code generation process. The generated back ends use tree pattern matching for code selection. Two methods for register allocation supporting a wide range of target architectures are provided. A general bottom–up pattern matching method avoids problems that occurred with previous systems using LR–parsing. The performance of compilers using generated back ends is comparable to very fast production compilers. Some figures are given about the results of using the system to generate the back end of a Modula–2 compiler.

## 1 Overview

Research on methods to formalize code generation has shown that describing code selection by pattern matching is very powerful. Machine instructions are described by tree patterns, and code selection is equivalent to finding a cover of the input tree using these patterns ([GFH82] gives an overview of early work in this area). However, this process is usually indeterministic, as there are several possible covers for a given input tree. Cost values associated with the instruction patterns are used to indicate desired qualities of the code, e.g. speed or code size. Ambiguities during selection of a cover are resolved in favor of the cheapest cover.

Several methods have been described for finding such a cover. Graham and Glanville [Glan78] proposed to use LR parsing. The collection of the instruction patterns written in prefix order is interpreted as a context–free grammar, and the cover is found by parsing the input tree (also linearized in prefix order) with a modified LALR(1) parser constructed from this grammar. Because the grammars are normally ambiguous, some heuristics and simplifications are built into the system to resolve the ambiguities. Several implementations of this method have been described, and some have been used in practical compilers [LJG82, GrHe84, GaFi85, JLHT83]. However, the heuristics and assumptions built into the system sometimes lead to surprising behavior, and detailed knowledge of the internal algorithms was desirable to write a specification that produced a good code generator.

A way to avoid this problem is to use a general pattern matching technique that guarantees that the best possible cover is selected. Among others, such general methods using dynamic programming algorithms have been used in the TWIG [AGT87] and the ASCOT [Jans85] systems. Our experience with the ASCOT system has shown that such a system is easy to use (i.e. it behaves like a naive user thinks it should) and that the specification of the instruction patterns is really independent of the actual pattern matching algorithm. On the other hand, the performance of the generated code generators was way short off the requirements for production compilers.

The BEG (Back End Generator) system presented in this paper uses a general pattern matcher and produces efficient code generators which can be used in practical compilers. The dynamic programming algorithm was modified such that no table interpretation is necessary. Instead, the algorithm is directly incorporated into the generated code generator, specially optimized for every operator of the intermediate language.

Besides code selection, register allocation is a major task of a code generator. The early code generator generators provided only minimal support for register allocation, and consequently the majority of errors found in the generated back ends was in this area. A specification for the BEG system includes a description of the register set of the target machine and the register

requirements of the instructions. Based on this information, two different register allocators can be generated together with the code selection. The on the fly method is very fast, but suitable only for machines with a regular register set, like the M68020. The other method allows to perform local register allocation for a wide range of processors including the 8086.

The BEG system is written in Modula–2. It translates a Code Generator Description (CGD) written in BEGL into Modula–2. We are planning a version which produces C. The BEG system was used to generate a code generator for Modula-2/MC68020 replacing the original code generator in the GMD Modula compiler Mocka. So we were able to compare the hand written and the automatically generated code generator.

This paper consists of 4 major parts. First we show how to describe code generators by tree patterns matching. Then an algorithm which determines minimal covers and a method for its efficient implementation are given. In Section 4 register allocation is described. Section 5 contains practical results including figures about using a code generator produced by BEG in a Modula–2 compiler.

# 2 Code Generation by Tree Pattern Matching

## 2.1 Introduction

We assume that earlier compiler phases have already translated the source program into an intermediate representation (IR). An IR–program is a sequence of expression trees. Each operator has a fixed arity, so we can write the trees in prefix order without parentheses. An operator may have attributes which are values known at compile time. Figure 1 shows an example IR–tree for a:=a+1 where a is supposed to be a local variable stored at offset 4 in the activation record whose address is returned by the BB (block base) operator. The cont operator returns the content of a memory location.

The code generator processes the IR one tree at a time and produces *abstract machine code* for it. 'Abstract' means that there exist abstract instructions which are mapped into several real instructions or into a call of a run time system routine.

A naive algorithm to translate a tree is to traverse it in postfix order and to produce code for each node separately. The resulting code, however, is very bad. The reason is that often one machine instruction can be used to implement several nodes of the IR–tree. For example the usual add instruction additionally performs an address calculation and a memory fetch.

Therefore the code generator has to combine several nodes to produce one machine instruction. To achieve this, pattern matching techniques have proven useful.

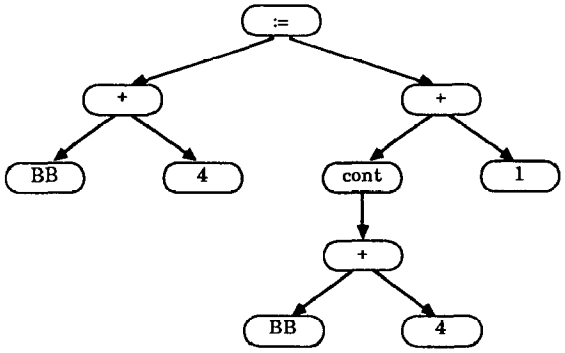



Figure 1: An IR–tree for a:=a+1

An instruction is described by a tree pattern. A tree pattern is like an IR–tree but may have *nonterminal* symbols as leaves. Each nonterminal in the pattern stands for an intermediate result calculated before. If the instruction returns a result the description also contains a *result nonterminal* indicating where the result is put into.

Figure 2 shows patterns and result nonterminals (here we need only one nonterminal named register) required to translate the tree of figure 1. The pattern for the add instruction may be read as:

> The Add instruction adds the content of a memory location (addressed by the sum of block base and a constant) and a register. It returns the result in a register.

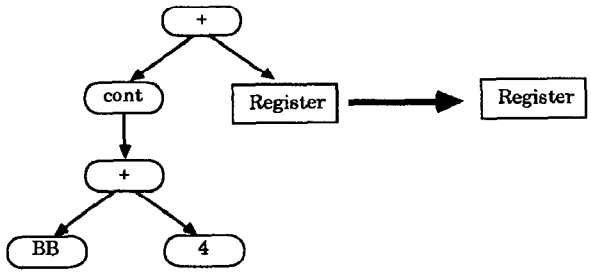
The rules of figure 2 may be read as tree rewriting rules. The input tree can be rewritten by them until it is totally consumed (assuming that rule 3 returns the empty tree). Usually instructions are generated as side effects of rule applications. We do not do this because we want to keep the tree structure, e.g. for subsequent register allocation.

Instead we smooth the nodes of the input tree which match against a single pattern together into a single tree node. This compound node is labeled with the corresponding rule. The resulting tree is called *cover tree* (see Section 2.3). Figure 3 contains an example.

It is also possible to view things the other way round by applying the rules backward (like grammar productions). This means pasting together of trees (with matching nonterminals). The problem the code generator solves is to paste the patterns together in a way that the original IR–tree results. The solution is represented in form of a cover tree.

The cover tree can be traversed in postfix order and code be produced for each node of the cover tree separately. That is done by the *output phase*. However, be-

Rule 1:

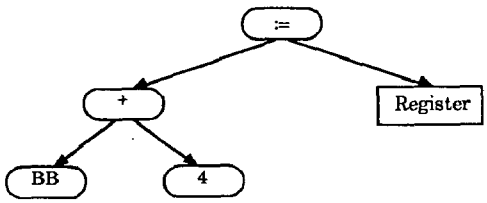



for A Register,4(BB) Cost 4

Rule 2:


for LA Register,1 Cost 3

Rule 3:




for ST Register,4(BB) Cost 4

Figure 2: Rules needed to translate the tree of figure 1

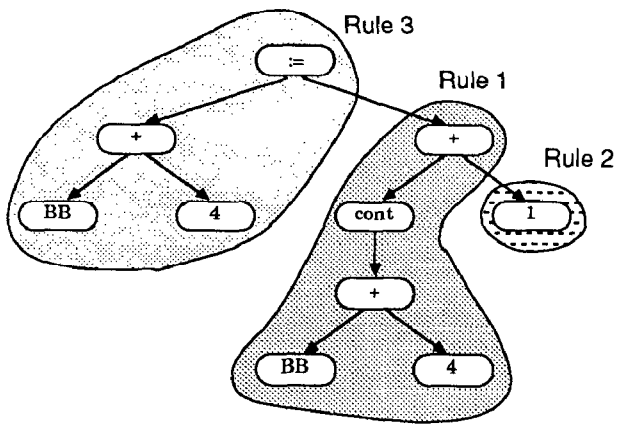



Figure 3: A cover tree

fore doing this the register allocator calculates some attributes of the cover tree required by the output phase.

Both methods to obtain cover trees are indeterministic and lead to a lot of different correct cover trees. A cover tree which corresponds to good code has to be selected. To measure the quality of a cover tree the user has to specify a cost value for each rule. The cost of a cover tree is the sum of the costs of all rules contained in it.

In section 3 an algorithm (and its efficient implementation) is described which always constructs (one of) the best covers (according to the cost measure of above). This is a great advantage in comparison to the LR parser based techniques. However, this advantage is not much better target code (in fact the quality of generated code is more or less equal to the code quality produced by good hand written or good LR parser based code generators) but simpler and easier understandable CGDs. So the development of CGDs becomes easier and faster.

The user who wants to write a CGD only has to know that the algorithm always finds a best cover. He does not have to deal with any details of the algorithm in order to achieve this (as might necessary with other methods).

An important condition for the correctness of a generated code generator is the *completeness* of the CGD. A CGD is complete if the resulting code generator can produce code for every possible IR-tree. With our approach it is sufficient if there exists a cover for every possible IR-tree.

To achieve completeness we first write a small but complete CGD. This CGD contains only simple rules and therefore would produce bad target code. Then we can add successively new rules to produce better code. Because BEG will find any existing cover, the completeness can not be lost by adding new rules. Hence this process leads to a complete CGD producing good code.

## 2.2 Further Description Techniques

BEG rules have the following form:

**RULE**
pattern [ '->' result nonterminal];
[**CONDITION** condition;]
**COST** integer_number;
[**EMIT** machine_instruction and/or
attribute evaluation;]

Example: The rule for the add instruction:

```
RULE
   Plus
        Content AddressPlus Register.a Constant
        Register
   → Register.r
   COST 4;
   CONDITION (0 ≤ Constant.value) AND
                    (Constant.value ≤ 4095);
   EMIT
     (* code to output
        A   r.register,Constant.value(a.register); *)
        ...
```

The pattern and the result nonterminal have been explained already. The result nonterminal must be present iff the root operator of the pattern calculates a result. Operators which have no result are called *top level operators*. Each input tree contains exactly one top level operator in its root.

After 'EMIT' an action is specified which may generate machine instructions or calculate some attributes. Actually this is just a piece of Modula code. It may contain statements to output assembler code or call an assembler module. The action is executed during the output phase.

The applicability of a rule may be restricted by a condition. The condition has to be a correct Modula condition.

Attributes of operators may be accessed in conditions and EMIT–actions. Additionally synthesized attributes may be computed during the output phase. They can only be accessed in EMIT–actions, not in conditions. We have experimented with a feature which allows to compute attributes during the pattern matching. The values of these attributes are available during condition evaluation. Unfortunately this destroys the optimality of the pattern matching algorithm. Fortunately this feature is not really necessary for our CGD.

The simple example of section 2.1 required only one nonterminal. In more complicated cases, however, we want to store the intermediate results in different places (e.g. different register classes) or in different forms (e.g. store the negative value). So we usually need a variety of nonterminals.

Nonterminals are also needed to describe addressing modes. Usually the machine allows several addressing modes. These can be used by a lot of different instructions. Using the technique as in the example above we would have to write a rule for each pair (instruction, addressing mode). To avoid this, usually some nonterminals are used. Section 4 contains an example about this.

When using several nonterminals *chain rules* become a very powerful feature. A rule is called chain rule if the pattern consists of a single nonterminal. Selecting the nonterminals and the chain rules is the crucial part of the design of a CGD. Section 5 contains a description of the nonterminals and chain productions of our CGD for MC68020.

### 2.3 Cover trees

The term cover tree has been informally presented in section 2.1. More precise a cover tree is a tree which fulfills the following conditions:

- The nodes are labeled with rule numbers.
- A node has as many sons as the pattern of the associated rule contains nonterminals.
- The i-th nonterminal of the rule of a node n is equal to the result nonterminal of the rule of the i-th son of n.

The *cost of a cover tree* is the sum of the costs of each node. The *cost of a node* is the cost of the rule associated with it.

We say a cover tree is a *cover* of an IR–tree t iff expanding the nodes of the cover tree results in t. Expanding a node n means replacing n by the pattern of the corresponding rule and replacing the i-th nonterminal in the pattern by the i-th son of n.

A cover tree is called a *minimal cover* of an IR–tree if no cheaper cover of the IR–tree exists. The following section describes an efficient algorithm for determining a minimal cover.

A cover tree t is called an *A–cover* (A is a nonterminal) of a tree, if it is a cover and there is a rule with result nonterminal A associated with the root of t.

## 3 Determining Minimal Covers

### 3.1 The Algorithm

To handle the rules of top level operators in an uniform way we introduce a new nonterminal Z and use it as result nonterminal. Now all rules have a result nonterminal. Our task is to determine a minimal Z–cover of the input tree. Therefore we use the following algorithm:

For every node $n$ of $t$ and every nonterminal $A$ the values $c_{n,A}$ and $r_{n,A}$ are computed. $c_{n,A}$ is the cost of a minimal A–cover of the subtree with root $n$. It is $\infty$ if none exists. $r_{n,A}$ is (the number of) the rule which is associated with the root of the covering tree. In case of $c_{n,A} = \infty$, $r_{n,A}$ is not defined.

$c_{n,A}$ and $r_{n,A}$ can be computed in a bottom up manner. To compute $c_{n,A}$ and $r_{n,A}$ of a node $n$ it is assumed that $c_{n',A'}$ and $r_{n',A'}$ for all descendants $n'$ of $n$ are available (that's trivially true for the leaves, so we can start there). We consider every non chain rule with the result nonterminal $A$ and with a pattern matching in $n$, i.e. the operators in the subtree with root $n$ and

the corresponding operators in the pattern are equal. Additionally the condition must be fulfilled.

The cost $C_r$ for the application of a rule $r$ is computed as follows. Suppose there occur $k$ nonterminals in the pattern. They are numbered $A_1 \ldots A_k$. Note that a nonterminal may occur more than once. According to the match there are descendants $n_1 \ldots n_k$ of n which match against the nonterminals $A_1 \ldots A_k$ respectively. Let $C_r = (\sum_{i=1}^{k} c_{n_i,A_i} +$ cost of rule r) and $r_0$ a rule where $C_r$ is minimal. Then let $r'_{n,A} = r_0$ and $c'_{n,A} = C_{r_0}$.

Then chain rules are processed. $c_{n,A}$ and $r_{n,A}$ are initially $c'_{n,A}$ and $r'_{n,A}$. As long as there is a chain rule r 'B $\rightarrow$ A' with cost $C$ and $c_{n,A} > C + c_{n,B}$ set $c_{n,A} := C + c_{n,B}$ and $r_{n,A} := r$.

Using the $r$ attribute it is possible to construct the cover tree. However it is not necessary to build it explicitly, it is sufficient to be able to traverse the tree. A traversal of the cover tree is performed by the register allocator and by the output phase.

To traverse the cover tree we start with the root of the IR–tree and the nonterminal $Z$. For processing a node $n$ with a given nonterminal $A$ we use the rule $r_{n,A}$. According to this rule $n_1 \ldots n_k$ and $A_1 \ldots A_k$ are defined as above. We then call the processing recursively for each $n_i$ with the nonterminal $A_i$.

A theoretical foundation of this algorithm can be found in [Emme88]. The theorem stating equivalence of indeterministic and deterministic finite automata was generalized leading to the algorithm above.

Figure 4 is shows the c and r values for the example of section 2. The $c$ values which are not given are $\infty$.

## 3.2 Implementation

### 3.2.1 Interface of the Code Generator

To increase efficiency we compute the attributes $c_{n,A}$ and $r_{n,A}$ in parallel to the building of the IR–tree. We assume that prior compiler tasks produce the code in postfix order. For every operator there exists one procedure in the code generator which builds the corresponding tree node and computes the $c_{n,A}$ and $r_{n,A}$ values. The procedure returns a pointer to the constructed tree. This pointer can be used as an operand for further operators.

```
PROCEDURE plus (op1,op2 : Toperand)
                : Toperand;
BEGIN
    new (node);
    WITH node↑ DO
        kind:=opplus;
        son[1]:=op1; son[2] := op2;
    END;
    compute c and r values of node
END;
```

BEG generates these procedures. The code to compute $c$ and $r$ is included inline, so no table interpretation is necessary.

### 3.2.2 Implementation of the Cost Values

Costs are implemented as integers. It is important not to use maxint for $\infty$. Instead we use maxint/$m$, because we frequently have to compute expressions like $c := \text{MIN}(a_1 + \ldots + a_k, c)$. If $k <= m$ we can implement this by

```
d:= a1+a2+ ...+ak;
IF d < c THEN c:=d;
```

For $m$ we use the maximum number of nonterminals occurring in one pattern.

### 3.2.3 Computation of $c_{n,A}$ and $r_{n,A}$

The $c$ and $r$ values are represented as arrays indexed by nonterminals. They are stored directly in the tree nodes:

```
Toperator = ( ⟨in CGD defined operators⟩ );
Tnode = ↑ Tnoderec; Toperand = Tnode;
Tnoderec = RECORD
    op : Toperator;
    son: ARRAY [1..maxarity] OF Tnode;
    costs : ARRAY Tnonterminal OF INTEGER;
    rules : ARRAY Tnonterminal OF Trulenr;
    ⟨user defined attributes⟩
END;
```

For an operator $o$ the code to compute the rules and costs arrays is generated in the following way. First the costs array is initialized with $\infty$. All rules with a pattern which starts with the operator $o$ are relevant. For every relevant rule $r$ with the result nonterminal $A$ the following code is generated:

```
IF ⟨operators match⟩ THEN
    c:= ⟨Cost_of_r⟩ + ⟨∑_{i=1}^{k} n_i ↑ .costs[A_i]⟩ ;
    IF node↑.costs [ ⟨A⟩ ] > c THEN
        IF ⟨condition of r⟩ THEN
            node↑.costs [ ⟨A⟩ ] := c;
            node↑.rules [ ⟨A⟩ ] := ⟨r⟩ ;
        END;
    END;
END;
```

To describe this in detail we need the term path($m$), where $m$ is a node of the pattern. For the root of the pattern the path is empty, for any other node $n$ of the pattern the path is p.son[i], if p is the path of the father $f$ and $n$ is the i-th son of $f$.

To check 'operators match' for each operator node $m$ in the pattern the predicate

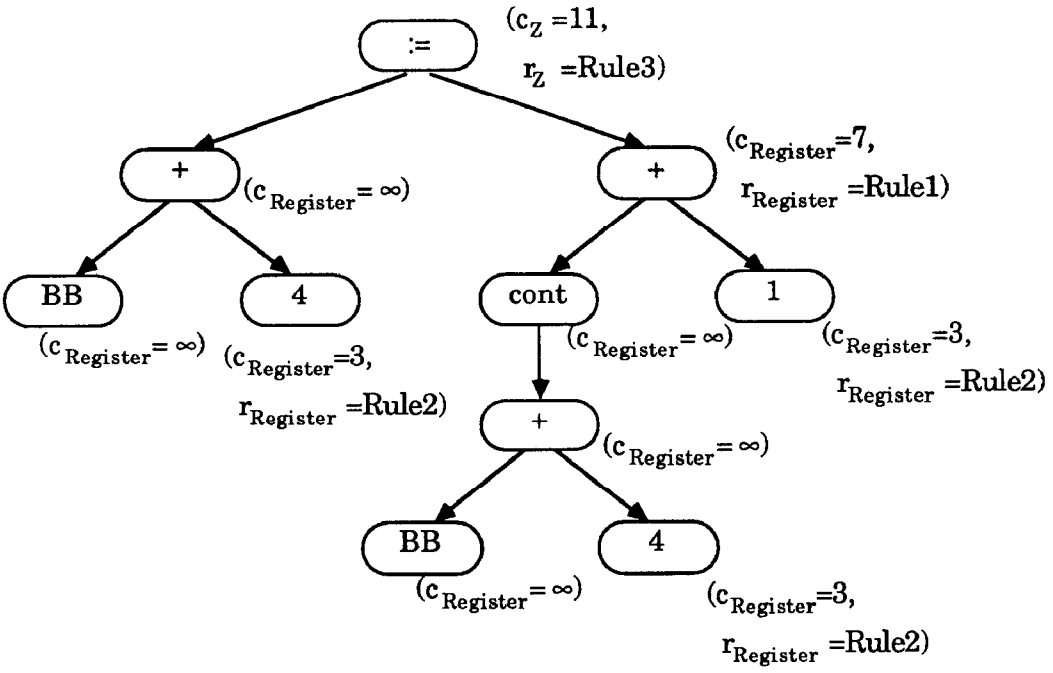



Figure 4: c and r for the example of section 2

```
IF node ↑.son[1]↑.op = opContent AND
   node ↑.son[1]↑.son[1].op = opAddressPlus AND
   node ↑.son[1]↑.son[1]↑.son[2].op = opConstant
   THEN
   c := 4 + node ↑.son[1].costs [ntRegister] +
     node ↑.son[1]↑.son[1].↑son[1]. costs [ntRegister];
   IF node↑.costs [ntRegister]>c THEN
      IF (0<=node↑.son[1]↑.son[1]↑
         .son[2]↑.atconstant.value) AND
         (node↑.son[1]↑.son[1]↑
         .son[2]↑.atconstant.value<=4095) THEN
         node↑.costs [ntRegister]:=c;
         node↑.rules [ntRegister]:=1;
      END;
   END;
END;
```

Figure 5: Code generated for rule in section 2

( node↑ ⟨path($m$)⟩.op= ⟨operator of m⟩ ) must hold. It need not be checked for the operator of the root. So the test can be omitted if there is only one operator in the pattern. For the calculation $\sum_{i=1}^{k} n_i \uparrow .costs[A_i]$ the following code is generated:

$$\begin{aligned} &\text{node} \uparrow \langle \text{path}(n_1) \rangle.\text{costs}[\langle A_i \rangle] \quad + \cdots \\ + \quad &\text{node} \uparrow \langle \text{path}(n_k) \rangle.\text{costs}[\langle A_k \rangle] \end{aligned}$$

Next we have to evaluate the condition. As a BEGL condition is Modula text it just has to be inserted here. It may include accesses to attributes, which have to be replaced by the correct path.

Finally the chain rules have to be considered. As chain rules are independent of the operator there is a separate procedure MatchChainRules for them. It is described in the next section.

The generated code contains many array accesses but the indexes are constant (figure 5 shows the code generated for the rule in section 2). It also contains a lot of common subexpressions. So it is very useful to compile it with an optimizing compiler.

### 3.2.4 Matching of chain rules

The code produced for a chain rule is very similar to that of ordinary rules. Actually it is much simpler. For a rule r '$B \rightarrow A$' with cost c the following code is generated:

```
IF node↑.costs [ ⟨A⟩ ] > ⟨c⟩ + node↑.costs [ ⟨B⟩ ]
THEN
   node↑.costs [ ⟨A⟩ ] := ⟨c⟩ + node↑.costs [ ⟨B⟩ ];
   node↑.rules [ ⟨A⟩ ] := ⟨r⟩ ;
END;
```

The application of a chain rule may depend on another chain rule. Therefore, chain rules have to be checked repeatedly until no chain rule is applicable any more. The simplest way is to put all the rules into a loop which stops if in the last iteration no rule has been applied. However this means that if only one rule is applicable all rules have to be checked twice.

As MatchChainRules is called very often (once for each operator), it is very interesting to improve it. BEG

does this by determination of a chain rules sequence (CRS) which guarantees that after its execution no rule is applicable any more. A CRS might contain some rules more than once. For example the MC68020–CGD contains 13 chain rules and the CRS generated by BEG 15. With this optimization the compiler became about 8% faster.

### 3.2.5 Traversal of the target tree

The register allocator as well as the output phase have to traverse the cover tree. Because it is possible to efficiently traverse the cover tree by using the data structure of the IR–tree and the $r$ attribute there is no need to build any additional data structures. Here we use the code produced for the output phase as example. BEG generates one procedure for each nonterminal A:

```
PROCEDURE emit_⟨A⟩
(VAR node : Tnode; VAR a : Userattributes);
BEGIN
    rule := node↑.rules [ ⟨A⟩ ];
    CASE rule OF
    ...
    END;
END emit_⟨A⟩;
```

For each rule with result nonterminal A the CASE contains the following code (for $n_i$ and $A_i$ see section 3.1):

```
emit_⟨A1⟩ (node↑ ⟨path(n1)⟩ , a1);
...
emit_⟨Ak⟩ (node↑ ⟨path(nk)⟩ , a⟨k⟩);
emit action of rule r
```

The type Userattributes is a record which contains the synthesized attributes which are computed by the emit actions during the output phase.

# 4 Register Allocation

## 4.1 Introduction

We consider it important to automatically build register allocators as well as code selectors. Writing them by hand is not only much work but often produces many errors difficult to find. Besides, a CGD already contains most of the information required for register allocation. A small extension allows the automatic generation of register allocators.

In the next two sections, it is explained how the information about the target machine is provided for the register allocator. Then the two register allocation methods supported by BEG are briefly described.

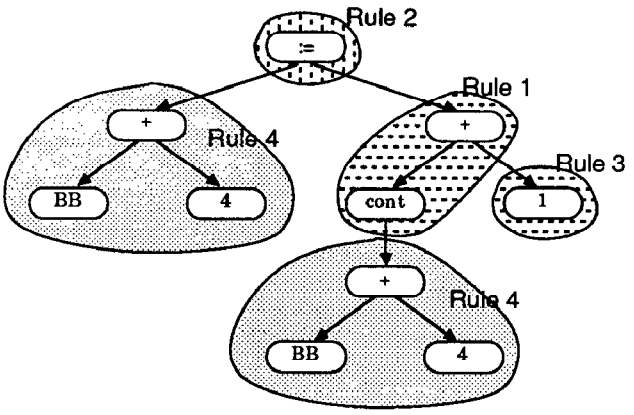



Figure 6: Cover tree with addressing mode rules

## 4.2 Pseudo Code Trees

Usually register allocation methods work on expression trees. Each node of the tree represents a machine instruction, which has several input and several (we allow only one) output registers. For example the add instruction with register operands is a binary operator as one would expect, but the add instruction of section 2 with one memory operand is only an unary operator because the memory operand does not concern the register allocator. If the memory operand had a base or index register these would be operands in the sense of the register allocator.

When we have constructed such an expression tree, we can use a normal register allocation method. Unfortunately the cover tree generally does not work (though it does in the example of section 2). The problem is the description of addressing modes. In order to avoid an exploding number of rules for all combinations of instructions and addressing modes, extra nonterminals for the latter are introduced. The instructions in section 2 might also be described by:

```
RULE 1
    Plus Content Address Register → Register;
    ...
RULE 2
    Assign Address Register;
    ...
RULE 3
    Constant → Register;
    ...
RULE 4
    AddressPlus BB Constant → Address
    ...
    EMIT
      (* calculate attributes of addressing mode *)
```

Figure 6 shows the cover tree according to the rules above. Not every node of this cover tree corresponds

to a machine instruction. The nodes labeled with rule 4 only compute some attributes used by their fathers. These attributes may even contain register numbers. So the register allocator can not handle these nodes like the other nodes.

Instead we transform the cover tree into a *pseudo code tree* prior to register allocation. Actually this transformation is only virtually, to get a better and easier description of the algorithms. No explicit data structure is build.

We divide the nonterminals into two classes: *register* and *addressing mode nonterminals*. In our example Register is a register nonterminal and Address an addressing mode nonterminal. The edges of the cover tree can be labeled accordingly. Then we combine all nodes which are connected by addressing mode edges into a single node. Figure 7 shows the resulting pseudo code tree of our example.

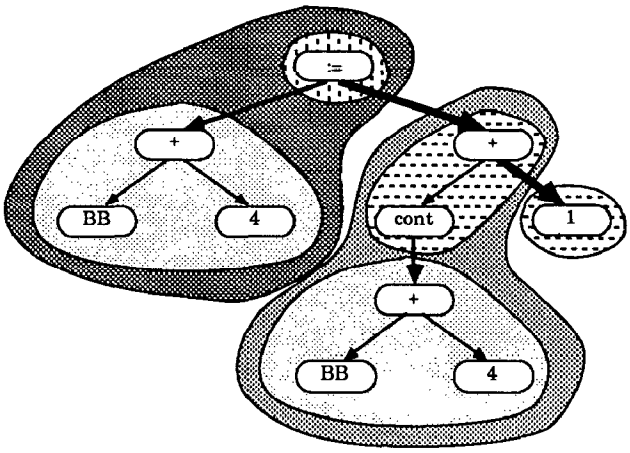



Figure 7: Pseudo code tree

Now the pseudo code tree could be transformed by changing the execution order of operands to improve the register requirements of the tree [ApSu87]. However this is not implemented yet.

## 4.3 Language Constructs of BEGL

One part of a CGD describes the register set of the target machine. In the simplest case all registers have to be enumerated. On some machines however several registers overlay each other. For example on the IBM/370 two 32-bit single registers may be used together as a 64-bit double register. We model this by considering the double register as a new register which overlays the two single registers.

In another part of the CGD the type of each nonterminal (register or addressing mode) must be defined. For each register nonterminal additionally a default register set must be specified.

For each operand and the result nonterminal of a rule the set of allowed registers can be specified. Therefore the registers are enumerated in brackets after the corresponding nonterminal. If no registers are specified the default register set of that nonterminal is used. The register allocator will only use the allowed registers for an operand or for the result. Sometimes register copy instructions have to be inserted to meet these constraints.

For every rule it is possible to specify the registers changed as a side effect of the emitted operation. To describe 2–address instructions one operand of a rule can be declared as the target operand. The same register is assigned for the result and for the target operand.

In other systems a fixed set of registers (or register class) is assigned to a nonterminal. We can do this by using the default register set. Additionally, further restrictions can be specified in the rules. This is useful for example to call a run time routine with parameters in registers, as the operands can be forced into fixed registers.

However, it seems better to solve the problem of different register classes by introducing nonterminals for each class, because the code selector does not consider the register copy operations inserted by the register allocator. On the other hand the code selector can not take into account if all registers of a certain class are used already and it would be better to use a register of another class instead.

It is the problem of the register allocation method to meet the specified constraints. This is not very difficult for a local register allocator, because is may freely insert register transfer instructions. It is, however, difficult to do it well, that means to insert only few instructions even under strong constraints. Therefore BEG supports two register allocation methods, a fast one, which does well only with few constraints and a slower one, which can handle even strong constraints. It is an open question for us if a good global register allocation method can be found which also can handle these constraints.

The register allocator imposes the following restrictions on the code generated by the emit actions:

- The result of the instruction must be returned in one register (but this register may consist of some other registers).
- An instruction may only change the result register and the registers explicitly specified as changed.
- After execution of an instruction (the code produced by a rule with a register nonterminal as result) all operand registers with the exception of the result register are reclaimed by the register allocator.
- One of the operand registers may also be used as result register.

Register nonterminals automatically have an attribute called register. The result of register allocation

is the calculation of this attribute. The attribute can be accessed during execution of the emit action. Additionally the register allocator may insert register copy, spill and reload instructions.

### 4.4 Register Allocation on the Fly

The on the fly register allocator is directly included in the procedures traversing the cover tree. Therefore the method is very fast. On the other hand it may produce bad code if there are too many constraints in the CGD. Processing a node involves the following steps:

- Check if the registers the operands are delivered in are suitable for the instruction (as specified in the CGD). If not, the operand has to be copied.
- All operand registers can be freed again.
- Select a register for the result.
- Insert spill instructions for every register other than operand or result registers which are alive during the instruction but changed by a side effects of the instruction (see change clause).
- Perform the emit action of the node.

The method works well if no look ahead is required. If there are many restrictions on operand registers (e.g. the operand has to be in an even register) or many registers changed by side effect of an instruction the method produces bad code or might even block. The difficulty is that when selecting a register the method does not take into account where the register will be used. So it might take a wrong decision which must be corrected by register copies or spill instructions later.

On the fly register allocation is sufficient for the MC68020, but it is not for the IBM/370 or the 8086.

### 4.5 The General Register Allocation Method

The allocator works on the pseudo code tree in a separate pass before the output phase. It processes the tree recursively and visits the sons of a particular node in reverse execution order. The recursive procedure gets a parameter, which contains a cost value for each register. It returns the register selected for the result and the set of registers changed by evaluation of the subtree. It tries to assign a register with minimal cost. If all registers have cost $\infty$ the result is mapped into memory or onto the stack.

When processing a node n for each son s the following cost values are assigned to a register r :

3 if the r is not changed between the calculation of the result of s and the execution of n.

2 if the r fulfills the condition above and if it is allowed (according to the CGD) for the corresponding operand of n.

1 the condition of cost 2 must hold, s must be the target operand of n and the cost c(r) assigned to the register r for the result of n is 1. However, if no register would get cost 1 the condition is relaxed to c(r)=2 and then c(r)=3.

$\infty$ all other registers.

Our implementation uses bit sets of registers, instead of ARRAY Register OF cost.
The ideas of this method are :

- The whole live range of a value is inspected before a register is assigned. So the constraints of the instructions defining and using the value can be considered. Side effects of instructions that change registers can be considered when selecting a register.
- Even when having 2–address instructions, all constraints on the register can be considered. For example, it is possible to have an instruction requiring one operand in a fixed register by specifying only one register as the set of allowable registers for that operand. This information is propagated down in the tree using cost 1.
- For short–lived values registers are assigned first. When we have registers of different quality (e.g. on the 8086 some instructions work with AX and all the other registers, but they are shorter using AX) we can select better registers first and achieve that they are used more often than others.

## 5 Practical Results

We have used BEG to generate another code generator for the already existing GMD Modula Compiler Mocka for the MC68020. Therefore we have written a CGD of 1625 lines containing 187 rules, 8 nonterminals and 99 intermediate operators.

Tables 1 and 2 show the nonterminals and some interesting rules of the CGD. The emit part of the rules normally contains Modula text which computes attributes or calls an assembler module to emit the desired code. Here we have simplified these parts. If a nonterminal or an operator occurs in a pattern more than once it gets a new name by suffixing it with '.name'.

The use of Modula in the emit parts leads to a great flexibility. So this was the right decision for experimental use. However, it is complicated to write the procedure calls for the assembler module necessary to emit instructions. So we are planning to allow an assembler like syntax for emit instructions.

**Dreg, Areg, Freg:** The value is contained in a data, address or floatingpoint register.

**Constant:** The value is statically known and given as an attribute.

**AregDispl:** The value is the sum of a statically known displacement and the content of an address register.

**AregIndex:** Attributes are areg, displ, scale and dreg. The value is content(areg) + displ + scale * content(dreg). scale may only be 1,2,4 or 8.

**Dest:** Dest has an attribute containing a complete MC68020–addressing mode. The value is the address calculated by this addressing mode (as the LEA–Instruction does).

**Ea:** Ea also has an attribute containing a complete addressing mode. The value is the content of the memory location denoted by the addressing mode.

Table 1: Nonterminals of the Modula/MC68020–Cgd

```
Areg → AregDispl;          EMIT AregDispl.displ := 0;
Areg → Dest;               EMIT construct addressing mode address register indirect
AregDispl → Dest;          EMIT construct addressing mode register indirect with displacement
AregIndex → Dest;          EMIT construct addressing mode register indirect with index
Const → Ea;                EMIT construct addressing mode immediate
Dreg → Ea;                 EMIT construct addressing mode register direct
Areg → Ea;                 EMIT construct addressing mode register direct
Ea → Areg;                 EMIT generate (MOVEA Ea.addressingmode, Areg.register);
Ea → Dreg;                 EMIT generate (MOVE Ea.addressingmode, Dreg.register);
Ea → Freg;                 EMIT generate (FMOVE Ea.addressingmode, Freg.register);
Dest → Areg;               EMIT generate (LEA Dest.addressingmode, Areg.register);
Content Dest → Ea;                       EMIT Ea.addressingmode := Dest.addressingmode;
AddOffset AregDispl.a → AregDispl.b;     EMIT b.displ := a.displ + AddOffset.value;
AddOffset AregIndex.a → AregIndex.b;     EMIT b.displ := a.displ + AddOffset.value;
Plus Dreg.a Ea → Dreg; TARGET a;         EMIT generate (ADD Ea.addressingmode,a.register);
```

Table 2: Some interesting rules of the Modula/MC68020–Cgd

| | Perm | Towers | Queens | Intmm | Mm | Puzzle | Quick | Tree | Bubble | FFT | Σ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mocka | 40 | 58 | 37 | 53 | 103 | 285 | 32 | 72 | 56 | 152 | 888 |
| Begra | 42 | 57 | 35 | 54 | 104 | 297 | 32 | 58 | 56 | 153 | 888 |
| Begfly | 42 | 57 | 34 | 54 | 102 | 299 | 33 | 56 | 56 | 151 | 884 |
| Sun | 67 | 90 | 28 | 83 | 101 | 263 | 41 | 81 | 63 | 150 | 967 |

Table 3: Comparison of code quality

| Compiler | compile time bench (1116 lines) | compile time CGPM (4539 lines) |
|---|---|---|
| Mocka | 2.9 sec | 13.0 sec |
| Begfly | 3.2 sec | 14.5 sec |
| Begra | 3.9 sec | 17.8 sec |
| Sun | 25.4 sec | not available |

Table 4: Comparison of compilation times

Our practical experience has shown that it is sometimes helpful to modify the IR prior to the pattern matching (see also [GrHe84]). Currently there is only a rudimentary feature implemented which will be improved in the future.

BEG needs 12 seconds to produce a code generator out of the CGD. Table 3 shows execution and Table 4 compilation times of a collection of bench mark programs (bench). Begfly denotes the Mocka compiler where the code generator was replaced by the automatically generated one using the 'on the fly' register allocator. Begra is like Begfly but using the general register allocation method. The code generator was translated with optimization. Sun is the Sun Modula compiler version 1.0. Table 4 also shows the compile time of the main module of the generated code generator implementing the pattern matching (CGPM). Due to differences in the predefined libraries, this module could not be compiled with the Sun Modula compiler. All measurements were carried out on a diskless Sun 3/60, all measured times are user times. The size of the code generator increased form 5140 lines (17117 tokens) for the hand written version to 9574 lines (27044 tokens).

The figures show the great differences between the two register allocators. The reason is besides the more complex algorithm of the general allocator the fact that the on the fly algorithm could be generated directly into the output phase, while the general allocator is a table interpreter working in its own pass. So it is desirable to use the on the fly register allocator when the target architecture allows it.

The execution time figures show no difference between the hand written and the generated code generators. The code produced is nearly of the same quality. We are planning to connect BEG with a global optimizer to further improve code quality.

The comparison with a production compiler shows that automatically generated code generators have become well suited for use in production compilers.

# 6 Conclusion

Automatic generation has proven a good method to build reliable and maintainable code generators in a fraction of the time of doing it by hand. Now with the techniques of BEG it is possible to really use this method in efficient production compilers. Extensions of the code generator description language may further improve the usability of BEG.

# 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 Karlsruhe, 1988

[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

[JaLa80] *H.-St. Jansohn, R. Landwehr:* CGSS: Ein System zur automatischen Erzeugung von Codegeneratoren. Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, 1980

[JLHT83] *H.-St. Jansohn, R. Landwehr, J. Hayek, M. Thätner:* Generating MC68000 Code for Ada. Proceedings of the 1983 ACM Conference on Personal and Small Computers, San Diego 1983

[LJG82] *R. Landwehr, H.-St. Jansohn, G. Goos:* Experience with an Automatic Code Generator Generator. Proceedings of the Sigplan 82 Symposium on Compiler Construction, Sigplan Notices, Vol. 17, Nr. 6

[Schr88a] *F.W. Schröer:* Das GMD Modula–2 Entwicklungssystem. GMD–Spiegel 1/88

[Schr88b] *F.W. Schröer:* Mobil: An Intermediate Language for Portable Optimizing Compilers. Internal report, GMD Forschungsstelle an der Universität Karlsruhe, 1988