Bottom up Tree Rewriting with MBURG: The MBURG Reference Manual

John Gough.*

Revision 0.7, September 1995

Abstract

This document describes the bottom up rewriter generator **mburg**.

mburg creates a hard-coded tree pattern matcher from a specified binary tree grammar. The grammar may be adapted to existing data structures by declaring the names of the types and fields which the pattern matcher must access. The output files are written in standard-conformant Modula-2.

An **mburg**-generated rewriter performs optimal rewriting, based on the declared costs of the supplied production-rules. It does this by performing dynamic programming at runtime, making two passes over the subject expression-tree.

All of the early experience with **mburg** has been derived from the use of the tool for code-selection. However, the tool is applicable to other tasks to which the theory of bottom up rewriting applies.

Copyright

Revision History

Version numbers of this document correspond to the version numbers of **mburg** which they describe.

Version 0.5, June 1995

The initial version.

Version 0.6, August 1995

Some minor corrections.

Version 0.5, September 1995

Incremental labelling implemented.

*This report is maintained in electronic form on the ftp server `ftp.fit.qut.edu.au` (Internet 131.181.2.16)

Contents

1 Introduction

One of the most general forms of computation known to computing science is *tree rewriting*. Since arbitrary syntactic structures may be represented by trees, the process of transforming one syntactic structure to another models many computational tasks.

In particular, the theory has found natural uses in language processing. In contemporary compiler theory, tree rewriting elegantly solves the problems of resolving type overloading, and optimal code selection for expression trees.[1]

Consider the three trees in figure 1. In this figure the leftmost tree is an abstract syntax

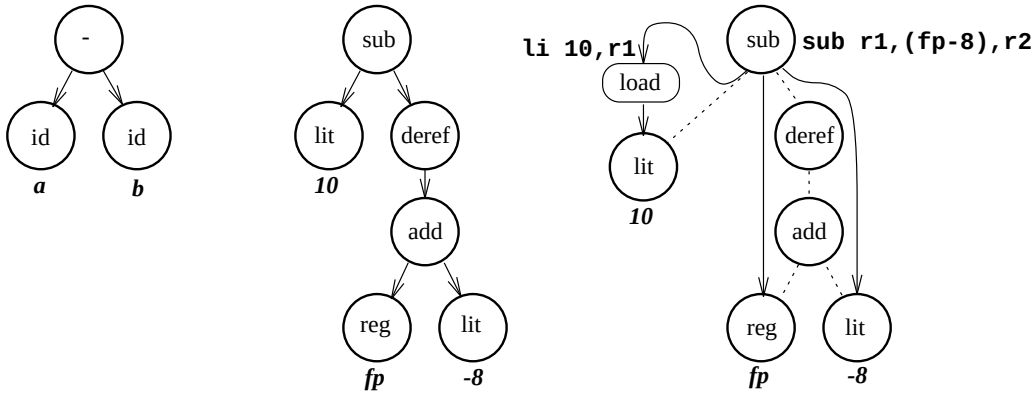

Figure 1: abstract syntax tree, code tree, instruction tree

tree arising from a source language expression such as "`(a-b)`". In the middle, is the result of static analysis. The names have been bound, and it has been discovered that the name a denotes a constant of value 10, while the name b denotes a local variable at an offset –8 from the frame-pointer fp. The middle tree represents the operations which describe the semantics of the expression.

The tree at the right shows a code sequence which implements the expression evaluation for a *CISC*-style machine which can take one of its operands from memory, and has an indexed address mode. Note that in the rewriting of the tree an extra node hase been introduced to signify the loading of the literal into a register, using the *load immediate*, "`li`" instruction. All four nodes in the right subtree have been subsumed into the address modes of the other instruction which subtracts the value in memory, at location (fp–8), from the value in the register. The result is placed in a register. In a *RISC*-style machine, the right subtree would have been loaded also, and the subtraction would be a register-from-register instruction.

Even within a particular instruction set, there are many, many ways of implementing even the simplest expression trees. These various *covers* of the tree will in general have different costs, as measured by some suitable metric. What we wish to do, with our tree rewriting, is to find the least costly implementation of trees, according to some cost metric. Furthermore, we wish to automatically construct tree rewriters from a specification for the tree-fragment patterns

[1] Although bottom-up tree rewriting is applicable to the resolution of type overloading, **mburg** cannot be used for the resolution of overloading in the case of type systems with user-defined overloading. **mburg** produces hard coded pattern matchers, which require that the alphabet of non-terminal forms is known at tool compilation time. In the case of user-defined overloading, the forms are not known until runtime of the tool.

which each instruction covers. For example, we would like to construct a tree rewriter for the example in figure 1 based on rules such as —

- subtractions may take a register, literal or memory operand on the right, but the left operand must be in a register
- literal values may be loaded into registers
- memory may be addressed using an index register and an integer offset

In practice, we seldom actually construct an explicit representation of the rewritten tree. Instead, once we have determined the optimal cover of the tree, we can let the rules guide the order of visits to the nodes. Thus, although our original trees are always binary we can always match patterns which have multiple arguments.

The theory of tree rewriting originated with Aho and Johnson[1], as long ago as 1976. It has become widely used only in the last few years. **mburg** is roughly comparable in its capabilities with **iburg**[2], although the implementation is significantly different.

It is possible to create faster tree rewriters using the so-called "burs-theory", although to take advantage of this potential speed increase requires a significant loss of flexibility, as in the **burg** tool.[3].

2 Bottom up tree rewriting

2.1 Productions and tree patterns

Each of the transformations in the tree rewriting is described by a production. The productions are comparable in their meaning to the productions of a context free grammar. In this case however, since the pattern which we wish to match is some arbitrary tree, it is necessary to adopt a syntactic notation capable of describing arbitrary trees.

2.2 Labelling and reducing

Bottom up tree rewriting is performed in two traversals over each *subject tree*, as we shall call the input structures to the algorithm.

The first traversal labels each node of the subject tree with the minimum cost of computing the tree rooted at that node into every possible *form* into which the value can be computed. Thus, in the case of a literal, leaf node, we might label the node with the cost of leaving the value as an immediate operand (probably zero), and compute the cost of loading the value into a machine register. All of these costs, and the index of the production which produced them, are stored in the *state vector* of the node. This pass is called *labelling*, and is performed bottom up, and left to right in the tree.

The second traversal uses the state information to choose the reductions to perform. Starting from the root of the tree, the production which produces the lowest cost for the goal symbol of the grammar is selected. This production will determine the *demanded form* for the subtrees of the node. In the subtrees, the demanded form will select the production which produces

that form at lowest cost, and the recursion will proceed. This traversal of the tree is called *reduction.*

As the reduction traversal unwinds, arbitrary semantic actions attached to each production can be performed. The placement of these actions is somewhat arbitrary, but in practice performing the actions *after* the recursion returns is best. Such a choice allows the actions to depend on synthesised attributes which are computed further down the trees during the reduction.

Note carefully that the first traversal is controlled by the syntactic structure of the subject tree, while the second is controlled by the semantic structure of the chosen productions. Thus a particular production may specify going directly from a node to one of its grandchildren, or may specify revisiting the same node, in the case of chain productions. In fact, the node visit order during reduction corresponds to the structure of the virtual, rewritten tree, shown in the right of figure 1.

The two traversals over the tree correspond to a dynamic programming attack on the cost minimization problem. In the case of **mburg**, the dynamic programming is performed at tree matching time. The **burg** tool, deals with a more restrictive class of specifications, and performs the dynamic programming at *matcher generation* time.[3] There is some evidence that the approach used by **mburg** may be a factor of about five slower than that of **burg**. However, against that must be weighed the loss of flexibility, and the relatively small proportion of the total compilation time used by code selection, in either case.

Early experience with **mburg** suggests that it is as fast as a shadow-stack automaton of the same pattern-matching complexity for some architectures. On other architectures, experience suggests that **mburg** is significantly slower that a comparable shadow-stack automaton.

2.3 Terminal and non-terminal symbols

mburg deals with two alphabets. The first of these, is the *non-terminal alphabet.* This is the set of symbols which appear as left-hand-sides of the productions. They are the various *forms* which may be computed as labels on the nodes of the rewritten tree. In the case of code selection, this alphabet will have a moderately small number of elements, such as the various forms of addresses, *Register, Immediate*, and perhaps a few others.

The other alphabet is the set of *terminal symbols*, which appear as labels in the nodes of the subject trees. These are so called because they appear as terminal symbols of the productions, and have no further expansion. Of course they may very well correspond to non-terminal symbols of the abstract syntax which created the tree, but they are terminal symbols of the rewriting grammar. In the case of code selection, this second alphabet will generally have a rather larger cardinality, with elements such as *and, add* and so on, corresponding to the various operations in the subject trees.

It is normally advantageous to make the terminal alphabet as large as possible, by splitting nodes which require different semantic actions into different terminal symbols. For example, *add* and *add with overflow check* should normally be separated. In order to see why this is the case, remember that the code emitted for these two cases is different. Furthermore, there are a large number of unchecked additions which arise within address computations, while checked additions arise only in explicit expressions from the source language. If the two nodes are merged, then the checked additions which occur in expressions will be needlessly checked

to see if they match patterns which can only occur in address expressions. Such checking for logically impossible patterns, will cause the labelling pass to run more slowly. The penalty which is paid for having a larger terminal alphabet is that the code size will increase slightly.

3 What Mburg Does

mburg takes a declarative specification of a tree-rewriting grammar, and produces a hard-coded pattern matcher and rewriter. The output files are written in ISO-conforming Modula-2. These files are usually combined with other source files, and compiled into the complete application.

mburg creates two modules, named *FormDefs* and *Match* as two definition-implementation pairs. The command line interface to the tool is dealt with later, in section 5. The overall dataflow is shown in figure 2. An input file is processed to create the two output modules,

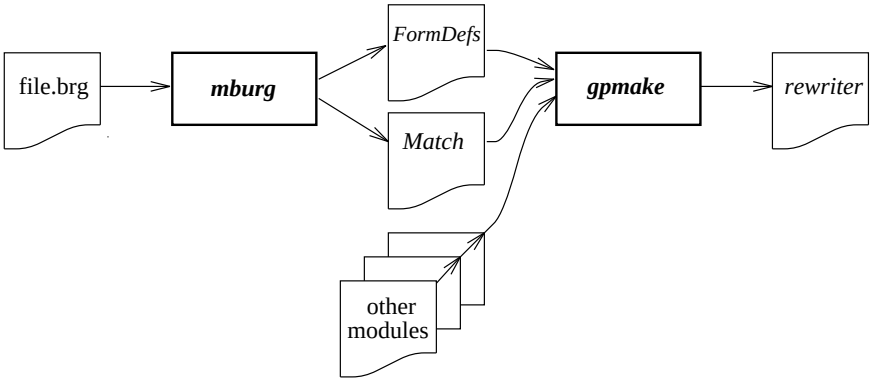

Figure 2: overall data flow in **mburg**

which are combined with other modules and compiled with a compiled with **gpm**, or with *gpmake*. The result is an output file, the name of which is determined in the usual way.

3.1 Interface to the tree

It is a restriction of the method, that the trees which are to be rewritten must be binary trees. Both nullary and unary nodes may exist in the tree, but unary trees must use the same selector for their single child as the binary nodes do for their left child.

The tree nodes must contain a single tag field of some ordinal type, and must contain a state field. The state field may be either an inline occurrence of the type *FormDefs.StateType*, or a pointer to this type. No other fields of the node type need be known to the tree-matcher, except as needed by the semantic actions attached to reduction rules.

The names of the tree node type and the module from which it is imported, together with the names of all the fields used by **mburg** must be declared in the input specification.

3.2 The application call interface

As well as the types which *FormDefs* and *Match* export, *Match* exports two procedures, which form the call interface to **mburg**. *LabelTree* implements the labelling traversal of the rewriter. It is called with a single parameter, which must be the root of the tree.

Reduce takes two parameters. The first is the root of the tree, and the second is the form to which the tree must be reduced. It is usually, but need not always be, the start symbol of the tree-grammar.

```
(*
 *  File automatically produced by Mburg from grammar <file.brg>
 *)
DEFINITION MODULE Match;
  IMPORT Types;
  FROM FormDefs IMPORT FormEnum;
  FROM NodeDefs IMPORT NodeDesc;

  CONST max = MAX(Types.Card16);

  PROCEDURE LabelTree (root : NodeDesc);
  PROCEDURE Reduce    (root : NodeDesc; goal : FormEnum);

END Match.
```

The procedure *LabelTree* performs labelling on the tree. In the current version this procedure dispatches a hardwired *matcher* procedure, using the terminal symbol of the root node as an index into an array of procedure variables. Each of these matcher procedures knows the arity of its own terminal symbol, and recursively dispatches matcher procedures on each child node, using the terminal symbols of each such node as index. After the recursions return from traversing the subtrees, the procedure sequentially attempts to match every declared pattern which starts with that terminal symbol. Each successful match checks to see if the new match is produced at a lower cost than previous matches which might have produced the same non-terminal form. Changes to the minimum costs are entered in the state vector of the node.

The procedure *Reduce* launches a recursive traversal at the root of the tree. In this case there is a separate procedure *ReduceNN* for each *production* in the grammar. These procedures call other *Reduce* procedures on the descendants of the current node, choosing successors based on the cost information computed into the state vectors during the earlier labelling traversal.

4 Input Specification

4.1 Input Syntax

The syntax of input to **mburg** is given by the extended Backus-Naur form in figure 3. The lexical rules are similar to those for Modula-2, that is, identifiers begin with an alphabetic character followed by an arbitrary number of alphanumeric characters. Character case is significant, and keywords are reserved.

Numbers are unsigned integers, and strings are enclosed in either single or double quote characters, and cannot contain either an end of line or the enclosing quote character.

```
      Mburg  →  "HEADER" {ANY} "DECLARATIONS"
                    ["INCREMENTAL"] {Declaration}
                    "RULES" {Rule} "ENDRULES" .
Declaration  →  "%start" NonTerm | "%OP" string | "%ENUM" string
             |  "%LEFT" string | "%RIGHT" string | "%STATE" string
             |  "%MNAME" string | "%TNAME" string | "%NEW" string
             |  "%ATTR" "<" {ANY} ">"
             |  "%FORM" "<" {ANY} ">"
             |  "%term { identifier "=" number } .
      Rule  →  NonTerm "=" Tree [PredFunc] [Cost] [SemAction] "." .
      Tree  →  identifier ["(" Tree ["," Tree] ")"] .
  PredFunc  →  "&" [identifier ] BalancedPar .
BalancedPar →  "(" {ANY | BalancedPar} ")" .
 SemAction  →  "(." [LocalDecs] ActBody ".)" .
 LocalDecs  →  "<" {ANY} ">" .
   ActBody  →  {ANY} .
      Cost  →  number .
   NonTerm  →  identifier .
```

Figure 3: EBNF for the **mburg** input

Comments follow the language *Ada* convention. That is, they begin with a double hyphen, and end with the end of line. This choice allows text which is meaningful to **mburg**, such as semantic actions for example, to contain strings which would be comments to Modula-2.

4.2 The header material

The header material is copied verbatim into the file "`match.mod`" and follows the other import statements. Thus the header material can contain both imports and any code which is needed either by the predicates or the semantic actions of the productions.

There are two different styles possible here. One is to place any required helper functions in some other module, and import those functions which are needed. The other style places such functions inline in the header.

4.3 Declarations

Declarations describe the interface between the generated pattern matcher, and the underlying tree data structure. There are sensible defaults for most of the defined symbols, with the possibility to override these defaults when necessary.

An optional first token declares the desired rewriter to perform incremental labelling of the tree during bottom-up tree-construction.

```
INCREMENTAL
```

The semantics of this choice are described in section 5.3.

The start symbol

The start or goal symbol of the grammar is declared with the syntax —

`%start` *non-term-identifier*

In the event that this declaration is missing, the left-hand-side symbol of the first production is taken to be the goal symbol.

Terminal symbol declarations

The terminal symbols of the grammar are declared with one or more terminal declarations. These have the syntax —

`%term` {*identifier* = *number*}

Zero is reserved for the error value.

The declared ordinal values must correspond to the values of the ordinal type declared with the operator-enumeration declaration. However, only those values of the enumeration which are actually used in the productions need to be declared in this section.

The nodetype and operator type declaration

The matcher module imports two types from some other module. One is the type of the node descriptors of the subject trees, and the other is the ordinal type which acts as a tag for these nodes. The names of these types and the module from which they are imported have sensible defaults. In the absence of any declarations the output file `match.mod` will contain the following declaration —

```
FROM NodeDefs IMPORT NodeDesc, OperEnum;
```

The name of the node descriptor type of the subject tree may be declared by the following syntax —

`%TNAME` *string*

The default name is "`NodeDesc`".

The name of the ordinal tag type of the subject tree may be declared by the following syntax —

`%ENUM` *string*

The default name is "`OperEnum`". This type is usually an operator enumeration, but needs only to be an ordinal type, with the zero value reserved for an error indication.

The module name declaration syntax is —

`%MNAME` *string*

which declares the name of the exporting module. The default name is "`NodeDefs`".

The node descriptor type must contain the tag field, and must also contain a matcher state field. This names of both these fields may be declared, but have simple default values. The type of the tag field is exported from the same module which exports the node descriptor type, but the type of the state field is depends on the types from the module *FormDefs* declared in the file "`formdefs.def`". This module declares a type "`StateType`", and the state field of the node descriptor type may either be this type, or a pointer to this type.

The node operator field

The nodes of the subject tree must have a field which contains the node tag. The default name for this field is "`op`", but the value may be overriden using the following syntax —

`%OP` *string*

The matcher state selector string

The nodes of the subject tree must have a field which contains (or points to) the matcher state. The default selector for the state is "`s^`", implying that the default is to access the state via a pointer field which is named "`s`". The state selector string is declared using the syntax —

`%STATE` *string*

State fields which are nested records will, of course, have declarations which do not involve an explicit dereference operator.

The left and right subtree fields

The subject trees have nodes which define binary trees, with two selectors which point to the left and right subtrees. These two fields may be either separate fields, or the two elements of an array. The default names of these two fields are "l" and "r" respectively, but these defaults may be overriden by the declarations —

`%LEFT` *string*

for the selector string which selects the left sub-tree, and —

`%RIGHT` *string*

for the selector string which selects the right sub-tree. Possible string forms are shown in the examples in section 4.3

The state allocation string

In the case that the state information is declared as a dynamically allocated area, it is usual to allocate this space during the labelling traversal of the tree. In this case, it is necessary to declare the storage allocation operation. This is done by the following declaration —

`%NEW` *string*

The string must have both the name of the state-allocation procedure, and also the arguments to the call. The name of the node to which the state is attached is always "`self`".

The default value for the state allocation string is the empty string, which is approriate where the state information is inline in the expression descriptor, and needs no separate allocation.

State attributes and imports

mburg automatically declares the attributes which it needs for labelling of trees, as a record type named "`StateType`" in the module *FormDefs*. In general extra attributes are needed for the semantic actions associated with the tree rewriting. These extra attributes are characteristic of the target tree structure, rather than of the source tree structure. Such attributes belong to the state type rather than the subject tree type.

Additional fields in the state-type record are declared with the following syntax —

`%ATTR` < $\{ANY\}$ >

All of the symbols which appear between the angle brackets "<" and ">" are copied verbatim, with appropriate indenting, into the declaration of the state type.

Since the angle symbols cannot occur in type declarations in Modula-2, this choice of delimiters is safe.

In general the extra attributes which are declared as part of the state type will be instances of externally defined types. Thus it is usually necessary to import these types into the header of the *FormDefs* definition. This is done with the following syntax —

`%FORM` < $\{ANY\}$ >

All of the symbols which appear between the angle brackets "<" and ">" are copied verbatim, with appropriate indenting, into the header of the *FormDefs* definition file.

Typical declaration examples

In the first example, figure 4, an existing tree data type has dynamically allocated state information added to it. The tree type has already defined names for the left and right subtree pointers, so these must be declared to **mburg**, so that the code which **mburg** produces can traverse these existing structures.

The second example, figure 5, is for a tree in which the state is inline in the node descriptor. The default names are used for most fields, except that the child (sub)trees are accessed by means of an array. In this example the declarations are explicit, for clarity, even when they do not modify the default. The second example also demonstrates the use of attribute declarations which add extra fields to the state type. In this case the state type declaration produced by **mburg** will be of the general form —

```
DECLARATIONS
    %start "Stmt"
    %MNAME "M2Designators"
    %TNAME "ExprDesc"
    %ENUM "ExprNodeClass"
    %LEFT "leftOp"
    %RIGHT "rightOp"
    %OP "exprClass"
    %STATE "state^" -- a pointer to StateType
    %NEW "ALLOCATE(self^.state,SIZE(StateType))"
```

Figure 4: Declaration example for existing tree

```
DECLARATIONS
    %MNAME "NodeDefs" -- same as default
    %TNAME "ExprDesc" -- same as default
    %ENUM "OperEnum" -- same as default
    %LEFT "kids[0]"
    %RIGHT "kids[1]"
    %OP "op" -- same as default
    %STATE "state" -- no dereference
    %ATTR < idx :  VRegister;
            hsh :  HashBucketType;
            off :  INTEGER; >
    %FORM < FROM NodeDefs IMPORT VRegister, HashBucketType; >
```

Figure 5: Declaration example with inline state

```
StateType =
  RECORD
    costs : ARRAY FormEnum OF Card16;
    rules : ARRAY FormEnum OF ProdIndex;
    idx : VRegister;
    hsh : HashBucketType;
    off : INTEGER;
  END;
```

where "`FormEnum`" is a type automatically generated by **mburg** for its internal workings.

4.4 Productions

The production rules of the tree grammar have a left-hand-side, which is a nonterminal symbol of the grammar, and a right-hand-side tree pattern. The tree pattern is followed by an optional predicate function or expression, and an optional cost. Finally, each production may have a semantic action associated with it.

Rule →NonTerm "`=`" Tree [PredFunc] [Cost] [SemAction] "."

Each production for a particular non-terminal symbol explicitly repeats the left-hand-side symbol, so there is no notion of alternation of patterns represented in the syntax.

Tree patterns

The tree patterns are specified in a fully parenthsized prefix form.

Tree →*identifier* ["(" Tree ["," Tree] ")"] .

Trees have a maximum arity of two, with the arity determined implicitly from the patterns. It is an error if all the productions of a particular terminal symbol do not imply the same arity.

Trees may consist of a single non-terminal symbol, or a tree pattern expressed in a parenthesized prefix form, beginning with a terminal symbol. For example the production rule —

```
Reg = add(Reg,Reg)
```

declares that the non-terminal symbol"Reg" matches a tree with an "`add`" node at the root, and with sub-trees which have the "Reg" form. Thus "`add`" is a binary node.

The production rule —

```
Reg = deref(Addr)
```

states that an unary "`deref`" node with a subtree of the "Addr" form matches the "Reg" non-terminal form.

The production rule —

```
Imm = literal
```

specifies that a nullary "`literal`" node matches the "Imm" non-terminal form.

Finally, the production rule —

```
Reg = Imm
```

specifies that a node of the "Imm" form, may be transformed into the "Reg" form. Such production rules, with a single non-terminal symbol on each side, are called **chain rules**.

Note that **mburg** is only able to distinguish between productions with nullary patterns, and chain rules by checking the kind of the identifier. In this case, terminal symbols must be declared explicitly, and every undeclared identifier is presumed to be a non-terminal symbol. **mburg** will give an error notification if such an implicitly declared non-terminal does not also appear as the left-hand-side symbol of at least one terminating production.

The recursive definition of the trees, permits nested patterns to be defined. Although most patterns refer only to the immediate children of a node it is possible to refer to grandchildren or even more distant offspring. The only limitation is that **mburg** does not allow a pattern to contain more than nine offspring. Nested patterns are often used in code selection applications. For example, the following might be a production for a machine which possesses an add-from-memory instruction, with a two-register memory address form —

```
Reg = add(Reg,deref(add(Reg,Reg))
```

The corresponding tree pattern is shown in the figure 6. Note that we do not care about

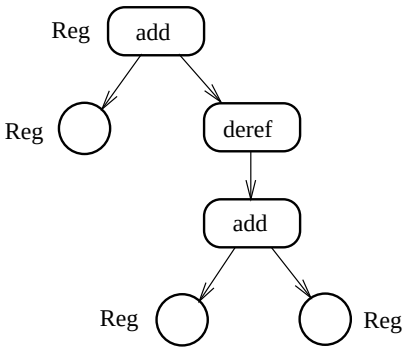

Figure 6: an example nested tree-pattern

the terminal tags of the leaf nodes of these patterns. We only care that the nodes have the required "Reg" form.

Predicate functions

Often particular patterns are only applicable if some semantic predicate applies to the node which is being matched. In some rewriter-generators this is handled by allowing the costs to be conditional expressions which evaluate to some "infinite" value in case that the predicate is not met. In **mburg**, as an alternative, predicates may be explicitly stated for each pattern, and costs must be constants.

$$\begin{array}{rcl} \text{PredFunc} & \rightarrow & \text{“\&”}\ [\textit{identifier}\ ]\ \text{BalancedPar}\ . \\ \text{BalancedPar} & \rightarrow & \text{“(”}\ \{ANY \mid \text{BalancedPar}\}\ \text{“)”}\ . \end{array}$$

Predicates are optional, and are introduced by the ampersand symbol "`&`" denoting that the production rule is only applied if the pattern is matched *and* the predicate evaluates to true.

There are two forms for the predicates. They may be Boolean function evaluations, or they may be arbitrary Boolean expressions enclosed in parentheses. Any parentheses within predicate expressions must occur in properly balanced pairs.

In general, predicates may refer to attributes of the current node or its children. The current node may be referred to as "`self`", but **mburg** also understands a macro notation which allows easy access to the leaf-arguments of the pattern, or to the fields of the state information.

The pointer to the N-th leaf-argument of the pattern is denoted by "`%`N" where $N \in [1..9]$, while self is denoted "`%0`". Thus, in the tree in the figure 6, we might have the following equivalences —

```
%0  =  self
%1  =  self^.leftOp
%2  =  self^.rightOp^.leftOp^.leftOp
%3  =  self^.rightOp^.leftOp^.rightOp
```

Note that "**P**ercent$N$" denotes the **P**ointer to the N-th leaf-argument node.

Access to the state attributes of the N-th leaf-argument of the pattern is denoted by "`$`N" where $N \in [1..9]$, while the state of self is denoted "`$0`". Thus, for the tree-pattern in figure 6, we might have the following equivalences —

```
$0.a  =  self^.state^.a
$1.a  =  self^.leftOp^.state^.a
$2.a  =  self^.rightOp^.leftOp^.leftOp^.state^.a
$3.a  =  self^.rightOp^.leftOp^.rightOp^.state^.a
```

where "a" is some attribute of nodes, and it is assumed that the state has been declared as being accessed by the string "`state^`". If access to the state is declared as requiring no indirection, then the expansions will automatically reflect this.

Note that "**$**$N$" denotes the **$**tate of the N-th leaf-argument node.

Using this notation, typical production rules which include predicate expresions might be —

```
Reg = mul(Reg,Imm)   &  IsPow2($2.iVal)
Reg = deref(fpAdr)   &  (Regvar($1.ofst) <> noReg)
```

We wish to perform all computations on the state of tree leaves at node initialization time, rather than at **mburg**'s runtime. In order to make this possible, it is a rule that neither leaf production rules, nor chain rules may have attached predicate expressions.

Cost values

mburg only allows numeric values as costs for productions. In the event that no cost is given, the cost is taken to be zero.

4.5 Semantic actions

Semantic actions are optional, and are enclosed in unique delimiters.

SemAction	→	"`(.`" [LocalDecs] ActBody "`.)`" .
LocalDecs	→	"`<`" $\{ANY\}$ "`>`" .
ActBody	→	$\{ANY\}$.

Local declarations are copied verbatim into the head of the reduction helper procedure for that procedure. The body of the semantic actions is copied verbatim into the procedure, following the automatically generated recursive calls to the reduction helper functions of the pattern leaves.

Non-leaf nodes

For non-leaf nodes, semantic actions are performed as part of the reduction traversal of the tree. Remember that matching patterns simply annotate a node as possibly being of the form designated by the rule's left-hand-side. If during the reduction traversal the rule is really selected, then the argument subtrees are visited and the semantic actions, if any, are executed.

As is the case for predicate functions, it is conventient to use the "%N" and "$$N$" symbolic notations to refer to the nodes and state attributes of the reduced tree. For example, a Sethi number state-attribute "`sethi`" might be computed by the following action —

```
Reg = add(Reg,Reg)              1 -- production cost is one
                    (.  IF $1.sethi < $2.sethi THEN $0.sethi := $2.sethi
                        ELSIF $1.sethi > $2.sethi THEN $0.sethi := $1.sethi
                        ELSE $0.sethi := $1.sethi + 1
                        END; .).
```

Leaf nodes

In the case of leaf nodes, it is clear that the pattern matching will always succeed, so it is possible to perform any semantic actions at node initialization time. In this way, leaf nodes of the rewritten tree do not need to be visited during the reduction traversal. The ability to move such semantic actions forward in time is fortuitous, since it allows semantic actions for leaf nodes to specify initialization of state attributes. This is particularly important in the case of dynamically allocated state workspace, since the predicates of other patterns may depend on such attributes being initialized.

Consider the productions —

```
Reg = mul(Reg,Imm)   &   IsPow2($2.iVal)                    1
                     (.  "shift instead of multiply" .).

Imm = literal                                               0
                     (.  $0.iVal := LitValueOf(%0); .).
```

In this example some information stored in the subject tree is extracted during the initialization of the state of the literal leaf node. This state information may then be queried by predicate expressions of production rules such as the strength-reduced multiply.

Semantic actions needing local variables

It is sometimes necessary for the semantic actions of productions to require the declaration of temporary variables. In this case, these must be declared at the head of the semantic action.

```
SemAction  →  "(." [LocalDecs] ActBody ".)" .
LocalDecs  →  "<" {ANY} ">" .
```

As an example, suppose that a particular production requires the emission of an overflow-checked instruction, followed by the allocation of a label, and the emission of a test and branch to that label. In this case, we shall need a local variable for the trap label.

```
Reg = addV(Reg,Reg)        1 -- add with overflow check
          (. <VAR trplab : LabelType;>
             EmitRRtoR(add,$1.reg,$2.reg,$0.reg);
             AllocateLabel(trplab);
             EmitLxxx(jo, trplab);     (* jump if overflow set *)
             ListTrap(trplab, linNum); (* add to the trap list *).).
```

The reduction helper procedure for this production will have the following form, where the default fieldnames have been assumed —

```
PROCEDURE ReduceNN(self :  Tree);
  VAR trplab : LabelType;
  VAR leaf1,leaf2 : Tree; (* declared automatically *)
BEGIN
 (* Prod NN <Reg: addV(Reg,Reg)> *)
  leaf1 := self^.l;
  leaf2 := self^.r;
  rHelp[leaf1^.s^.rules[Reg]](leaf1);
  rHelp[leaf2^.s^.rules[Reg]](leaf2);
 (* semantic actions follow ... *)
  NewVReg(self^.s^.reg);
  EmitRRtoR(addcc,leaf1^.s^.reg,leaf2^.s^.reg,self^.s^.reg);
  AllocateLabel(trplab);
  EmitLxxx(jo, trplab);     (* jump if overflow set *)
  ListTrap(trplab, linNum); (* add to the trap list *)
END ReduceNN;
```

where *rHelp*[ ] is the array of procedure variables of which *ReduceNN* is one element. The calls to these two procedure variables perform the recursive reduction of the two subtrees, prior to the semantic actions being executed.

As an alternative, in this case the real semantic action could have been abstracted away into a separate, helper procedure. For more complicated cases, the abstracting away of the actions into procedures is often a sensible strategy. Moving complicated actions out of the specification prevents the production rules being cluttered up by lengthy semantic action declarations, and may allow semantic actions to be shared between several productions.

5 Running Mburg

mburg is invoked by name from the command line. If the program is invoked with no arguments it prints the following usage prompt —

```
[grange]/wrk/mburg> mburg
# Mburg generator-generator version 0.7, Tue Oct 24 19:37:26 1995
# Mburg usage: mburg [options] filename<cr>
# Options ---
#       -p ident        -- use ident as name prefix
#       -v              -- verbose mode
#       -t              -- emit tracing code
# Input is <filename[.brg]>
# Output files are ---
#       [prefix]Match.(def,mod)
#       [prefix]FormDefs.(def,mod)
[grange]/wrk/mburg>
```

mburg is normally invoked with just one filename argument, with optional extension "`brg`". There are a small number of options.

5.1 Input and output files

Normally, **mburg** creates two modules — *FormDefs* and *Match*, in the four files — "`formdefs.def, formdefs.mod, match.def, match.mod`". The program also creates a listing file, which contains important information in the event that any error is discovered during

processing. The listing file has the same name as the input file, but with filename-extension "`lst`".

5.2 Command line options

Name prefixes

It is possible to specify a prefix on the command line, which is prepended to each of the module names. Using this facility, it is possible to avoid name clashes, if several tree rewriters exist in the same program.

Thus, the command —

```
mburg -p Sparc sparc.brg
```

would create the modules — *SparcFormDefs* and *SparcMatch*, in the files — "`sparcformdefs.def, sparcformdefs.mod, sparcmatch.def, sparcmatch.mod`".[2]

Verbose mode

mburg normally does its work silently, but has an optional verbose mode. If **mburg** is invoked with the **–v** option, then it chatters on about the progress of its internal processing.

```
[grange]/wrk/mburg> mburg -v sparc
# Mburg generator-generator version 0.7, Tue Oct 24 19:37:26 1995
# Mburg: parsing
# Mburg: parsed correctly
# Mburg: checking grammar
# Mburg: creating matcher file <match.mod>
# Mburg: creating matcher file <match.def>
# Mburg: creating formdefs file <formdefs.mod>
# Mburg: creating formdefs file <formdefs.def>
# Mburg: writing header block <651> bytes
[grange]/wrk/mburg>
```

Tracing mode

The **–t** option makes **mburg** produce code which gives a message for each pattern which it matches. This mode is so verbose that it is of little use except for debugging productions. Typically, it may be used after changes to the rule-set, to check that the new rules are being exercised, and are behaving as intended.

Let us suppose that a file "`foobar.brg`" is the pattern-matcher specification file for a program with program module named *CodeGen*. The following commands generate a tracing mode pattern matcher, and compile that into the program.

[2] **mburg** uses the standard module `gpfiles` to generate filenames from the module names. Thus the case of the filenames is controlled by the setting of the *GPNAMES* environment variable. The examples here assume that this variable has the value "`lower`".

```
[grange]/wrk/mburg> mburg -t foobar
[grange]/wrk/mburg> gpmake codegen
## compiling formdefs.def ...
## compiling match.def ...
## compiling formdefs.mod ...
## compiling match.mod ...
## building codegen ...
[grange]/wrk/mburg>
```

Now, when the compiled program *CodeGen* is invoked, each of the matching procedures of module *Match* give a cost and a diagnostic for each direct pattern which is matched during each labelling traversal. Each time a pattern is matched, the automaton prints out the accumulated cost of applying that production, together with the production (and the predicate, if any). At each reduction a marker is emitted, which delimits the matches of separate labelling traversals. The reduction marker also prints the ordinal of the goal non-terminal symbol for that particular reduction.

As an example, consider the single line procedure —

```
PROCEDURE Zap(VAR a : TwoDArray; i,j : IndexType);
BEGIN
  a[i,j] := 3;
END Zap;
```

This has the code tree shown in figure 7. In this figure, the leaf nodes are shown shaded. The

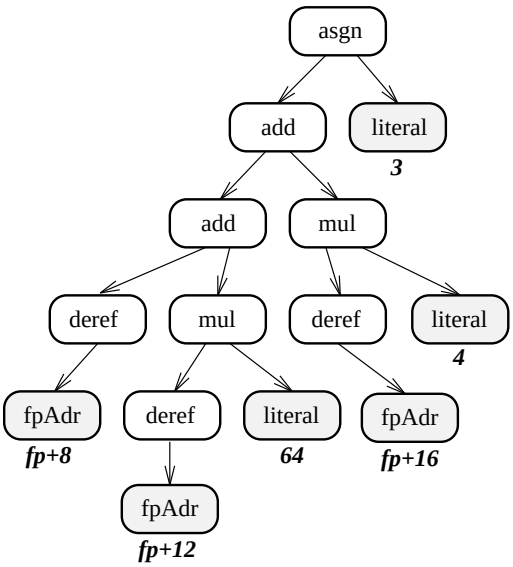

Figure 7: assignment to an array element

figure contains implicit information about the array sizes and frame offsets of the parameters in the procedure. For machines with multiply instructions, the whole body of this procedure is a single expression.

Running the tracing version of *CodeGen* over this tree, with a typical set of productions, might lead to the following output to standard-out —

```
[grange]/wrk/mburg> codegen zap.d
  1         <Reg = deref(Adr)>                        match 1
  1           <Reg = deref(Adr)>                      match 2
  6         <Reg = mul(Reg,Reg)>                      match 3
  2         <Reg = mul(Reg,Imm)> & IsPow2(self^.r^.s^.iVal)
  4       <Reg = add(Reg,Reg)>                        match 5
  1         <Reg = deref(Adr)>                        match 6
  6       <Reg = mul(Reg,Reg)>                        match 7
  2       <Reg = mul(Reg,Imm)> & IsPow2(self^.r^.s^.iVal)
  7     <Reg = add(Reg,Reg)>                          match 9
  9   <Stmt = assign(Adr,Reg)>                        match 10
  8   <Stmt = assign(add(Reg,Reg),Reg)>               match 11
==== Reduce to NT #1====
[grange]/wrk/mburg>
```

The matches are performed bottom-up, left to right in the tree, with indenting corresponding to the depth in the tree at which the root of the pattern is matched.

A limitation of the information which is given by the current release, is that there is no indication of matches at leaf nodes, since these are fixed, and require no pattern matching. Furthermore, only the *primary* match at each node is given since, once again, the chain productions which flow from each match are fixed.

In this example, match 1 is at the bottom left "*deref*" node of the tree, and has cost 1. Note that the match $Adr = fpAdr$ at the leaf node is not shown.

The subtree rooted at the bottom left "*mul*" node has a left tree which matches the same dereference production for a cost of one. The multiply node itself matches two productions. In the first (match 3) the two subtrees may be placed in registers, this has a cost of one for the left dereference, one to load the literal on the right with the chain production $Reg = Imm$ and a cost of four for the multiply, giving a total of six. However at the same node the strength-reducing pattern in match 4 has a cost of one for the left dereference, zero for the immediate literal on the right, and a cost of just one for the shift which replaces the multiply.

The root of the whole tree has two matches. The first of these, match 10, makes use of the chain production $Adr = Reg$ which allows register values to be used as addresses. In this case the complete address would be loaded for cost 7 (by match 9) with a further cost of one to load the literal on the right, plus one for the assignment. The second match folds the addition of the two address components, which are thus accessed at a total cost of six (by match 5 and match 8) with a further cost of one to load the literal, plus one for the assignment.

5.3 Incremental labelling

If the code tree is built bottom-up, then the labelling may be done incrementally as the tree is constructed. This is a considerable saving, since no separate labelling traveral of the tree is required. The optional keyword *INCREMENTAL* in the specification indicates to **mburg** that an incremental labeller is to be produced.

Recall that in the classical labelling algorithm a recursive traversal of the tree labels the nodes bottom up as the depth-first recursion unwinds. In this case, for each tree, a single call of `Label(`*root*`)` is made, and the *Label* procedure launches the depth first recursion. In the case of non-incremental matchers created by **mburg**, the actual labelling is done by an array of dispatched procedures which are indexed on the terminal symbol of each node.

In the incremental case, each node of the tree is created, and has its subtrees attached. A (now non-recursive) labelling procedure is selected according to the terminal symbol of the node which has just been created. This procedure is then called with the new node as argument. Each of these labelling procedures relies on the root nodes of its subtrees having been already labelled.

6 Error Messages

6.1 Syntactic and file errors

If **mburg** detects any syntactic errors in its source file, the single message "Incorrect source" is printed to the standard error stream. It is usually necessary to check the output listing file to determine the exact nature of the error. In the listing file, there is a visual indication of the exact position at which the error is detected, together with some information on the nature of the error.

mburg uses a recursive descent parser, constructed by the program **coco/r**, with all of the limitations that are implied in the error recovery capabilities of this underlying technology.

If the input file cannot be found, or any of the output files cannot be created, then **mburg** issues an error message to the standard error stream, and then terminates. In the event that further arguments follow the filename, then the warning is issued —

`#Mburg: arguments after <`*filename*`> ignored`

and processing continues.

6.2 Semantic errors

Semantic errors write a message to the list file, but also write a diagnostic to the standard error stream. The following errors are detected —

error 50: `chain production of symbol <`*identifier*`> cannot be conditional`

error 50: `leaf production of symbol <`*identifier*`> cannot be conditional`

As described at the end of section 4.4, leaf productions and chain productions cannot have a predicate attached to them.

error 51: `terminal symbol <`*identifier*`> is undeclared`

The nominated symbol is known to be a terminal symbol, as it appears as the prefix of a non-nullary tree, but is undeclared.

error 100: `terminal symbol <`*identifier*`> already defined`

The indicated symbol is doubly declared.

error 101: `terminal symbol <`*identifier* `> _and_`
`terminal symbol <`*identifier*`> have same value`

Two symbols have been declared with the same ordinal value.

error 102: `terminal symbol <`*identifier*`> must have a positive value`

Every terminal symbol must be defined with an ordinal value which is non-zero and positive.

error 103: `terminal symbol <`*identifier*`> has contradictory arity`

Two productions for the indicated symbol give it different arity.

error 104: `non-terminal symbol <`*identifier*`> has no productions`

The given non-terminal symbol has no productions. It is also possible to get this error when a leaf terminal symbol is undeclared.

error 105: `non-terminal symbol <`*identifier*`> cannot be reached`

There are no productions which lead from the declared start or goal symbol, which produce the indicated non-terminal symbol.

error 106: `non-terminal symbol <`*identifier*`> does not terminate`

There are no productions for this symbol for which every symbol in the right-hand-side pattern can be shown to be terminating. This usually means that there are missing productions in the grammar, or possibly an identifier name has been mis-spelt.

7 Hints for use

There are two circumstances which are common in the application of **mburg**. In the first, the traversals are performed on a tree which is constructed especially for rewriting. In such a case, the design of the node type may be chosen to be convenient to the rewriting tool.

However, the other circumstance arises when the structure of the subject trees has its design dictated by other considerations. For example, there may be already be a rich variety of attributes which exist for purposes of static semantic analysis. In such a case, it is normal to not require all of the tree nodes to permanently provide space for the state vector, but rather to dynamically allocate state workspace just during the rewriting.

Where this is the case, any attributes of leaves which are needed for predicates of the pattern matching, or for semantic actions, should be written into the tree nodes during state initialization. Consider the value of a literal. This will be obtainable from the subject tree node, but it is simplest to assume that the treematcher does not have to understand too much of the structure of the subject tree nodes. It may be convenient to encapsulate all knowledge of such things into a value-returning function, which is called during initialization.

```
Imm = literal                    0 -- zero cost
            (. $0.litVal := GetLitValue(%0);.).
```

In this way the semantic actions are separated from the details of the subject tree data types. Remember also that semantic actions for nullary nodes, that is, leaf nodes of the grammar, are performed at initialization time rather than at reduction time.

A particular issue arises for the implementation of state attributes for nodes which may be connected by chain productions. Consider the chain production —

```
Reg = Imm                       1 -- unit cost
            (. NewVReg($0.dstReg);
               EmitItoR($1.litVal,$0.dstReg);.).
```

Since this is a chain production, the nodes %0 and %1 are actually the same node. If the state vector is implemented by a variant record, clearly the fields *dstReg, litVal* must not overlap in storage. Indeed, the use of variant records for the state vector is theoretically problematical, since the semantic actions of chain productions treat what is in reality the same record as being of two different variants simultaneously. Although **gpm** will not prevent the use of unsafe accesses in this way, it is probably bad design.

Finally, there is a small limitation in the use of bottom up tree rewriting to effect constant folding. Consider the plausible production —

```
Imm = add(Imm,Imm)              0 -- zero runtime cost
            (. $0.litVal := $1.litVal+$2.litVal;.).
```

This seems unexceptionable, but the add node may be the right operand of a binary operation which has a range-limiting predicate —

```
Reg = add(Reg,Imm)   & FitsInImmed($2.litVal)  1
            (. NewVReg($0.dstReg);
               EmitRItoR($1.dstReg,$2.litVal,$0.dstReg);.).
```

In this case, the folding will not take place until the reduction traversal, but the predicate needs to be evaluated during the labelling travesal. The plausible notion simply does not work, in cases where the value of a folded literal is used by a predicate during labelling.

An example of a circumstance in which a tree is reduced to a form other than the start symbol may be helpful to consider. Suppose the "tree" from which code is to be generated is inherently a directed acyclic graph (*DAG*). This may happen for statements such as *INC* and *DEC* in Modula-2. In each of these cases, the shared subexpression is the address of the first parameter variable, see figure 8. In all such cases of shared expressions, it is possible

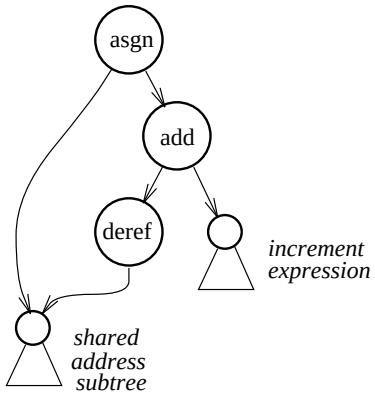

Figure 8: directed acyclic graph for *INC*(lVal,rVal)

to declare a local variable to hold the shared value, and to "un-dag" the structure into an asssignment which places the value in the variable, followed by the (now) tree which uses the value.

Unfortunately this strategy does not always lead to the best code, since it may be wasteful to compute a very simple value into a register. The notion of allowing the frontend to determine when to declare a local variable is also flawed, as this decision is target dependent, and hence properly belongs in the code generator.

An alternative strategy to consider is to define two new forms, *Temp* and *VReg*. A *VReg* is a register value, which is however guaranteed to be a unique virtual register, with just a single definition. A suitable production might be —

```
VReg =  Reg          -- move to a virtual register if necessary
        (.<VAR new : VRegister;>
           IF IsPhysicalReg($1.dstReg) THEN
             NewVReg(new);
             EmitRtoR(mov,$1.dstReg,new);
             $0.dstReg := new;
           END;
           %0^.code := regLeaf;.).
```

In the case that the value is in a physical register, it is copied to a new virtual register, otherwise it is left unchanged. In either case the node is overwritten as a register leaf-node.

The *Temp* form is an aggregate for any value which is safe against modification by other expression evaluations, in whatever form is least costly for that expression. Typically we might have —

```
Temp= VReg.                    -- any virtual register, no cost
Temp=  fpAdr.                  -- a constant local address,
Temp=  NameA.                  -- any non-indexed address
Temp=  Imm.                    -- any immediate
```

Of course the particular forms that a temporary might take are dependent on the target architecture.[3]

It is now possible, when a shared node is encountered, to execute —

```
LabelTree(node);
Reduce(node, Temp);
```

The resulting root node is able to be used as a shared node in later trees, since it is guaranteed to not require emission of any code.

7.1 Reclaiming memory

The "`%NEW`" declaration allows dynamically allocated state vectors to be produced as part of the labelling traversal. Sometimes there is a need to deallocate such state incrementally after reduction.

In order to deallocate space, it is necessary to place a procedure in the header which explicitly deallocates the tree which has been reduced. In order to facilitate the writing of such procedures, **mburg** defines an array "`arity[...]`" which holds the arity of each terminal symbol. A typical deallocation procedure would be —

[3] Note that a synthetic production might need to be added to ensure that the non-terminal symbol *Temp* is reachable from the start symbol.

```
PROCEDURE Dispose(this : NodeDesc);
BEGIN
  IF arity[this^.op] >= 1 THEN
    Dispose(this^.l);
    IF arity[this^.op] = 2 THEN Dispose(this^.r) END
  END;
  DISPOSE(this^.s);
END Dispose;
```

All that remains is to ensure that this procedure is called after each reduction has completed. This may be done by manually inserting a call to *Dispose* as the last line of *Reduce*, or by placing calls as part of the declared semantic action of every reduction to the start symbol.

References

[1] A V Aho and S C Johnson. Optimal code generation for expression trees, *Journal of the ACM* Vol 23(3), 488–501, 1976.

[2] C W Fraser and D R Hanson, and T A Proebsting. Engineering a simple efficient code-generator generator, *Letters on Programming Languages and Systems*, Vol 1(3), 213–226, 1992.

[3] C W Fraser and R R Henry, and T A Proebsting. BURG — Fast, optimal instruction selection and tree-parsing, *SIGPLAN Notices* Vol 27(4) 68–76, 1992.