Twig Reference Manual

Steven W.K. Tjiang

Twig is a language for manipulating trees. A *twig* program consists of a set of pattern-action rules together with associated declarations. Patterns describe trees to be matched. Actions calculate costs, perform tree manipulations and other functions such as emitting code. A *twig* program is translated by the `twig` compiler into subroutines and tables in a host language. In the current implementation, the host language is C.

A *twig* program manipulates trees by first finding a minimum cost covering of the input tree. The actions of the rules whose pattern parts composes the covering is then executed. The minimum cost covering is determined using dynamic programming. This technique naturally resolves many ambiguities that may be in the specifications.

The prime purpose of *twig* is to create tree manipulation programs. One interesting application of tree manipulation is code generation and *twig* has been used to implement a code generator for the pcc2 compiler on the VAX.

Contents

1 Introduction **2**

2 Motivation **2**

3 The *Twig* Language **5**
3.1 Lexical Issues and Conventions 5
3.2 Rules 6
3.3 Tree Patterns 6
3.4 Costs 7
3.5 Trees 7
3.6 Prologue and Inserts 8
3.7 Declarations 8
3.8 Costs and Action code 9

4 Pattern Matcher Operation **9**
4.1 The Costing Phase 10
4.2 The Execution Phase 10

5 Some Examples **11**
5.1 Expression Trees to Prefix 11
5.2 Short Circuited Boolean Evaluation 14
5.3 Code Generation 14

6 How to run `Twig` **14**

7 Organization of the `Twig` Compiler **16**

8 Performance **19**

1 Introduction

Twig is a language for writing tree manipulation programs. A *twig* program does this by using a matcher to find a minimal cost covering of a tree. The covering is composed of user supplied templates. Trees are manipulated by executing the actions associated with the templates of their coverings. These actions may rewrite the tree, update other data structures, or generate output.

Figure 1 gives an overview of the *twig* system. The round boxes in the figure indicates files and the sharp cornered boxes are processing programs. The user writes the *twig* specification file. This file is then compiled by the `twig` compiler into host language files. In the current version of *twig*, the host language is C. The host language files are then compiled and linked with other user supplied files. The resulting program represented by the large box at the bottom of figure 1 is the tree manipulation program. The details of this box will be discussed later.

2 Motivation

Although *Twig* is a language for manipulating trees it was originally motivated by pattern directed code generation. In many compilers, the typical structure is a *front end* and a *back end*. The front end reads the input source program, checks its syntax and contextual conditions, and builds an intermediate representation of the program. The language used to do this representation is often called the *intermediate representation* and will be abbreviated as IR. The back end takes the IR and translates it into the target machine code.

That isn't the whole story but it's close. Both ends may be sets of programs rather than one program. There may also be a *middle* that performs transformations on the IR. The intention of these transformations is to improve the target code that comes out of the compiler. For this discussion, we can ignore the middle.

IRs can vary greatly from compiler to compiler. However the IRs are implemented, they are encodings of graphs. Usually three types of graphs are evident[4]. There is a flowgraph whose nodes are *basic blocks* and the edges corresponds to transfer of control between the basic blocks. Basic blocks are not featureless. They are graphs themselves. Some compilers will represent them as a forest of *directed acyclic graphs* or DAGs. The nodes are data values and operations. Since operations yields values, one can think of nodes as corresponding to values. An edge exists from node a to node b if and only if the value corresponding to a depends on the value corresponding to b. Other compilers will use trees instead of DAGs. Trees are not as general as DAGs and these compilers overcome that by using some type of labelling or copying.

Writing a front end usually involves writing a recognizer for the source language, designing a symbol table, and determining an error recovery strategy. What has made this process easy are parser generators like YACC[8]. They separate out the task of recognizing the source language from the other details. This makes the task intellectually more manageable. The

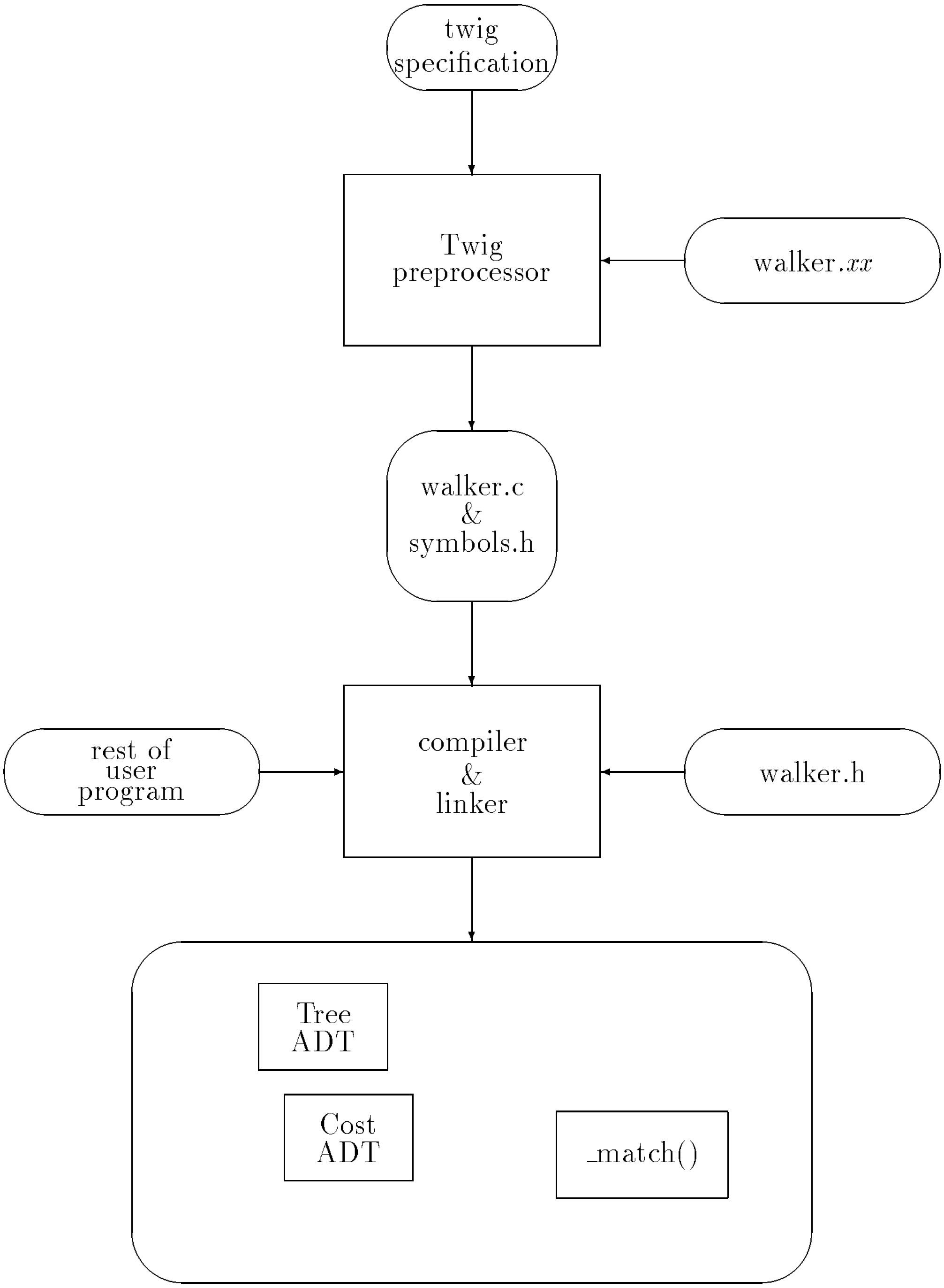


Figure 1: The **Twig** system

compiler implementor now specifies the source language via grammar productions. Symbol table manipulations, and IR construction rules are attached to productions and activated when a source language construct is recognized. The front end is then *automatically* generated by processing the specification with a parser generator. The result is a high quality front end whose performance is close to that of a hand-crafted front end.

Providing the same type of facilities for automatically building back ends has been much more difficult. Extending current front end parsers to do back ends has been workable but not entirely successful. The basic idea is that the back end implementor writes a *machine grammar* for the IR and code generation occurs as actions that are executed when patterns are found in the IR[5, 6]. Unfortunately, these grammars are large and ambiguous. Using parsers to handle them have usually met with many combinatorial problems [6]. Furthermore, these code generators are slow although their output is often of high quality. The software engineering advantages one see with parsers in the front end are not so evident in this circumstance. Instead of having to deal with the complexities of code generation, the implementor now has to deal with the complexities of the very large machine grammar.

Another approach at automatically building back ends is to start with known good algorithms for generating code. From these algorithms, we can build an abstract interpreter and then specifications can be written to drive them. We hope that by doing this we can derive a much more natural code generation specification technique. *Twig* is one possible result of this approach. The algorithms that form the basis of *twig* are presented in [3] and [7]. *Twig* specifications are much less complex than parser based specifications. Ambiguities are handled naturally and do not complicate the specifications.

In [3], it is shown how optimal code can be generated for an expression tree given a description of a machine's instruction set. This description is in term of tree templates. The algorithm first determines a minimal cost covering of an expression tree using dynamic programming. The covering is composed from the templates in the description. The code generated is then the instructions represented by the templates in the covering. This algorithm was shown to generate the shortest code sequences for an abstract machine and its running time was shown to be linear in the number of nodes in the expression tree in [3].

This is all very well in theory but what about in practice. Code length is not always the quantity that one wishes to minimize. One might wish to minimize the number of memory accesses or machine cycles executed by the generated code. Fortunately, the algorithm can also minimize these quantities. In fact, it can minimize any strictly increasing cost function. Let I denote a machine instruction and $I_0 I_1 I_2 ... I_n$ denote a sequence of instructions. Then a cost function f is strictly increasing if and only if $f(I_0 I_1 I_2 ... I_n) < f(I_0 I_1 I_2 ... I_n I_{n+1})$ for all instruction sequences $I_0 I_1 I_2 ... I_n$ and instruction I_{n+1}. Another shortcoming in [3] is that no efficient algorithm is given for matching the templates in the tree. This is where [7] comes in. In that paper, two algorithms are presented for tree matching. *Twig* combines the work presented in the two papers.

Twig differs from the algorithm as presented in [3] in the following ways:

- It provides a convenient specification language for the templates.

- Tree templates in [3] were labelled with either *Register* or *Memory*. This label being the storage class that the instruction corresponding to the template would put a result. *Twig* generalize tree labels so that they can be any symbol.

- Instead of associating an instruction with each template, *twig* associates an action.

- In [3], code is generated by a bottom up traversal of the minimum cost covering. *Twig* allows a top down traversal of the cover and also the ability for the actions to rewrite parts of the tree.

- The algorithm in [3] determines the evaluation order which gives the optimal code. *Twig* does not do this. The evaluation order determination was omitted based on the conjecture that most of the time, simple left to right evaluation of the leaves will yield near optimal results. In situations where a nonstandard evaluation order is required, a new *twig* pattern could be added that codes the order explicitly (see Section 4.2).

In this manual, a brief discussion will be given of the *twig* language and how *twig* programs can be incorporated with C.

Many statements have been made in this manual without proof. If the reader wishes to pursue it further, more details about the algorithms will be presented in an upcoming paper, [2].

3 The *Twig* Language

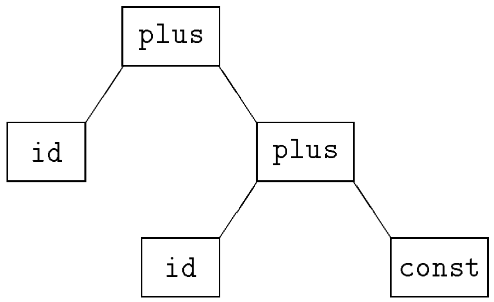
3.1 Lexical Issues and Conventions

Currently the following identifiers are reserved keywords of *twig*:

action	cost	insert
label	node	prologue

;:(),= are special characters. All blanks, tabs, formfeeds and newlines are ignored by *twig* but they may not appear inside identifiers and numbers. Identifiers are nonempty strings made of letters, digits or underscores and starting with a letter. Numbers are nonempty string of digits. Code fragments or action parts are enclosed in braces. Inside code fragments, C lexical rules must be obeyed except that strings of the form $...$ that are not inside C strings have special meaning to *twig*. In the following sections, *id* denotes an identifier; int_1, int_2 denotes numbers; *ccode* denotes C code fragments; ... indicates repetition of the previous item; [...] indicates that ... is optional.

The input to the *twig* program will be referred to as the subject tree.



`plus(id, plus(id, const))`

Figure 2: A Tree and its prefixed form

3.2 Rules

The fundamental unit of a *twig* program is a rule.

$$label\_id : tree\_pattern \quad [\ cost\ ]\ [\ = action\ ]$$

Intuitively, the pattern is used to match a subtree. The *cost* code fragment is then evaluated. The resulting cost is recorded by the matcher for use in dynamic programming. The *action* is executed if the rule is part of the least cost cover of the tree. The details of pattern matching will be discussed in Section 4.

If the *cost* part is missing, `twig` will insert default code that returns the special value, `DEFAULT_COST`. A missing *action* part indicates that nothing will be done when a match is found.

3.3 Tree Patterns

Tree patterns are specified in prefix parenthesized form and can be described by the following BNF:

$$tree\_pattern \rightarrow node\_id \mid label\_id$$
$$tree\_pattern \rightarrow node\_id\ (\ tree\_list\ )$$
$$tree\_list \rightarrow tree\ ,\ tree\_list \mid tree$$

Figure 2 gives an example of a tree pattern and its prefix parenthesized form.

There are two types of symbols: *node_ids* and *label_ids*. *Node_ids* are used to denote internal nodes and leaves. *Label_ids* label tree patterns and are analogous to nonterminals in context free grammars. For example, the following *twig* rules without their action parts describe simple expression trees with the `plus` operator.

```
expr:  plus(expr, expr)
expr:  identifier
expr:  constant
```

Here, `identifier` and `constant` are *node_ids* representing leaves, and `plus` is a *node_id* representing an internal node whereas `expr` is a *label_id*. Pattern leaves that are *label_ids* are called *labelled leaves*. In the first rule, both leaves of the pattern are labelled.

`Twig` associates an integer with each node symbol and label symbol. These integers are used by the *twig* pattern matcher as encodings for the symbols. As the matcher traverses the tree, a user supplied subroutine is called to provide an integer corresponding to each node.

3.4 Costs

To increase the flexibility of representing costs, the tree matcher views costs as an *abstract data type* or ADT. For example, costs may be represented as an integer or as a vector of integers with each element representing the cost of a specific resource. A cost ADT suitable for *twig* must have the following four definitions:

- `COST` is a C type. Although the proper functioning of the tree matcher does not depend on the internal details of the `COST` type, it must have the type information for storage allocation purposes.
- `INFINITY` is the maximum attainable cost value. The matcher uses `INFINITY` to initialize its data structures.
- `DEFAULT_COST` is the cost value returned by rules without a cost part.
- `COSTLESS` is a function of two cost values. It must return true if and only if the first argument is less than the second.

3.5 Trees

As with costs, trees are treated as an ADT. Here, using an ADT is even more important because trees come in a variety of shapes and representations. *Twig* would be overly complicated if it had to know any more than the fundamental properties of trees. Thus, *twig* treats trees as an acyclic directed graph of almost featureless nodes with one distinguished node, the root. Each node has only one feature and that is an integer corresponding to the *node_ids* discussed above. To provide this view to *twig*, the user must provide the following definitions and subroutines.

- `NODEPTR` is the type of a node. The actual details of the `NODEPTR` are irrelevant and `twig` uses this definition only for storage allocation and declaration purposes.

- `NULL` is used to denote a null `NODEPTR` value. In the current implementation this is the same `NULL` used by the C standard I/O package and need not be defined by the user. This restricts the `NODEPTR` representation to be a pointer.

- `mtGetNodes`(r, n) returns the nth child of r where r is a `NODEPTR`. The leftmost child is `mtGetNodes`$(r, 1)$. If $n >$ degree(r) then the function should return `NULL`. `Twig` expects the expression `mtGetNodes(NULL`$, 0)$ to denote the root node of the subject tree.

- `mtValue`(r) returns the integer associated with r (See section 3.3).

- `mtSetNodes`(r, n, c) replaces the nth child of r with c. This routine may be used to replace whole subtrees with others. `mtSetNodes(NULL`$, 0, c)$ is used by `twig` to set the root of the subject tree to c.

3.6 Prologue and Inserts

`prologue` *ccode*;

signals to `twig` that *ccode* should be inserted at the beginning of the output source file. *Ccode* should contain definitions relevant to the C code in rules that are defined later in the *twig* source file.

`insert` *ccode*;

causes *ccode* to be placed into the source file. There can be multiple inserts and they will be placed into the source file in the order that they appear.

3.7 Declarations

All *twig* identifiers are declared before they are used.

`node` $id[(int_1)][= int_2]\ldots;$

A node declaration declares one or more identifier to be associated with nodes of the subject tree. The identifiers are assigned numbers by `twig` but the user can overide the assigned number by specifying int_2. The degree of the node identifer, the number of children, is assumed to be fixed. Normally, `twig` will deduce the degree when *id* is used in a rule. However, the user may explicitly give the degree by specifying int_1.

`label` $id\ldots;$

A label declaration indicates that the following *id*'s are to be used as labels of rules.

3.8 Costs and Action code

Code fragments such as the *cost* and *action* clauses of a rule are specfied by enclosing C code in braces. All legal C constructs are permitted in code fragments. In addition, the following are provided for convenient access to the subject tree and internal data structures of the matcher.

- $\$\%n\$$ denotes a pointer value to the matcher data structure for the nth labelled leaf. Thus to access the cost value associated with that leaf, the notation $\$\%n\$$ $\rightarrow$`cost` may be used.

- $\$\$$ denotes a pointer value to the root of the subject tree.

- $\$n_1.n_2.n_3.\ldots.n_{k-1}.n_k\$$ denotes a pointer value to the n_kth child of the $n_{k-1}th$ child of the n_{k-2}th child of ... of the n_1th child of the root of the subject tree. Each n_i is a positive non-zero integer.

Some special constructs are available to code fragments in the cost part of the specification. The statement "`ABORT;`" when encountered during the execution of the cost code, signals to the matcher that this pattern is to be rejected. When a "`REWRITE;`" statement is encountered, control is returned to the matcher and the rule will become a *rewrite* mode match. When the end of the cost code fragment is reached, control is returned to the matcher and the rule becomes a *defer* mode match. The statement "`TOPDOWN;`" is like "`REWRITE;`" except that the mode of the match becomes *topdown*. The meanings of these modes will be explained in Section 4.

Cost values are returned to the matcher by assigning to the "`cost`" variable in the cost clause. If no assignment is made to the `cost` variable then the returned cost will be `DEFAULT_COST`.

Action clauses are expected to return a new tree which will replace the subject tree. This is done by returning using the "`return(`*new_tree*`);`" statement. where *new_tree* is of type `NODEPTR`. If execution reaches the end of the action clause, the matcher resumes execution and the subject tree is not modified.

4 Pattern Matcher Operation

The pattern matcher operates in two phases: the costing phase and the execution phase. During the costing phase, a minimal cost covering of the subject tree is found. The execution phase invokes the actions that are associated with the patterns making up the covering. Execution phases may begin during the costing phase to execute the covering of a subtree as described in Section 4.2.

```
e:  plus(e,e)
e:  plus(e, plus(e,e))
e:  id
e:  const
```

$S \rightarrow e$
$e \rightarrow plus(e, e)$
$e \rightarrow plus(e, plus(e, e)$
$e \rightarrow id$
$e \rightarrow const$

Figure 3: Pattern and Grammar

4.1 The Costing Phase

Let the patterns of the *twig* specifications be interpreted as grammar productions with *label_ids* as non-terminals and *node_ids* as terminals. If we add a production $S \rightarrow$ *label_id* for all *label_ids* where S is the start symbol, then a covering for a tree is analogous to a derivation for the prefixed form of the tree. Figure 3 shows some patterns and its corresponding grammar. The productions of the form $S \rightarrow$ *label_id* reflect the fact that any *label_id* may start a cover.

The matcher finds the least cost cover by doing a preorder traversal of the subject tree. At the same time, it builds a skeleton tree that is structurally isomorphic to the subject tree. When a match is discovered the cost clause of the pattern is invoked to calculate the cost. Many patterns with different labels could match at any given node but only the lowest cost pattern for each label is recorded in the skeleton.

When a pattern is matched, its label is then used as input to the pattern matcher so that matching of patterns with labelled leaves can begin. This process is analogous to a reduce action in bottom up parsers.

4.2 The Execution Phase

The execution phase starts when the costing phase is complete or when a sufficiently low cost *rewrite* mode rule is encountered. Let M be a matching rule and $L_1, L_2, L_3, ..., L_n$ be the labelled leaves of the matching pattern. The following procedure is used to execute the action parts of M.

Procedure *Execute*

- If M is a *deferred* mode match then execution occurs by first applying *Execute* to the L_is from left to right. That is, in the order that they appear in the prefix form of the tree. When all the L_is are executed, the action part of M is invoked.
- If M is a *rewrite* mode match then just the action part of M is executed. When the action part returns the matcher deletes the skeleton corresponding to the subtree that M matched and rescans the new subtree that may have been substituted by executing M.
- If M is a *topdown* mode match then only the action part of M is executed. To execute a labelled leaves, L_i, M's action part may do so explicitly by using the `tDO($%i$)` macro. This allows the user to arbitrary order the execution of the labelled leaves.

When the costing phase is over, the execution phase is started by picking the lowest cost match at the root of the subject tree. This will be the root of the minimum cost cover. The match is executed as described above.

The execution phase may begin before the costing phase is over. This occurs when a rewrite rule matches and its cost is lower than all other matches at that subtree. In this situation the rewrite rule is executed immediately and the new rewritten subtree is scanned once more. The prescence of rewrite rules throws a wrench in the theoretical niceties of the matching algorithm. For example, there is now no guarantee that the algorithm will terminate because the tree can be repeatedly enlarged. Rewrite rules can be dangerous. They can be triggered unintentionally if the user is not careful to abort them in situations where they are not wanted. However, if used with care they can help to reduce the size of the *twig* specification.

5 Some Examples

5.1 Expression Trees to Prefix

A *twig* program to print out the prefix form of expression parse trees is shown in Figure 4.

- The rules do not have cost calculations. Since there are no ambiguities costs are not necessary.
- The second rule is a topdown mode rule. This is essential in printing the prefix form. If it was omitted the postfix form would be printed.
- Before any matching can begin `_matchinit` must be called. Trees can then be processed by calling `_match` after arranging that the call `mtGetNodes(NULL, 0)` returns the root of the subject tree.

```
prologue        {

typedef         struct node *           NODEPTR;
#define         COST                    int
#define         INFINITY                100000
#define         DEFAULT_COST            0

NODEPTR Root;
struct code     {
                char op;                /* null if node is a leaf */
                NODEPTR left, right;
                char *id;
};
};

node            nOp nIdent;
label           lExpr;

lExpr:          nIdent
                = { printf("%s", id); };
lExpr:          nOp(lExpr,lExpr)
                { TOPDOWN;}
                = {
                                putchar($$ →op);
                                tDO($%1$);
                                tDO($%2$);
                };

insert          {

mtValue(root)
                NODEPTR root;
{
                if(root→op==0)
                                return(nIdent);
                else
                                return(nOp);
}
```

Figure 4: Printing Expression trees to Prefix form

```
mtGetNodes(r,n)
        NODEPTR r;
        int n;
{
        if(n==1)
                return(r→left);
        else if(n==1)
                return(r→right);
        else if(n==0 && r==NULL)
                return(Root);
        else return(NULL);
}
mtSetNodes(r,n, c)
        NODEPTR r, c;
        int n;
{
        if(n==1)
                r→left = c;
        else if(n==2)
                r→right = c;
        else if(n==0)
                Root = c;
}
main(argc,argv)                    /* called by user only */
        char **argv;
{
        _matchinit();              /* initialize the matcher */
        ... get a tree and set Root to it
        _match();                  /* do the match */
}
}
```

Figure 4: (cont.) Printing Expression trees to Prefix form

5.2 Short Circuited Boolean Evaluation

Short circuited boolean evaluation is naturally topdown. A fragment of *twig* program that implements this is shown in Figure 5.

- Labels for true and false branches are passed down to rules below by putting them into the nodes. Another way of accomplishing this is to use an auxillary argument stack.

- The rules assume that every test generates a branch to both the false and true label. On conventional machines, extra branches can be eliminated by recognizing that one could generate code that falls through to its true or false labels. This can be done by separating *lTest* into *lTrueTest* and *lFalseTest* which branches only when true and false respectively. Rules can then be written to take advantage of this.

5.3 Code Generation

Figure 6 is an example of *twig* program that can be used to generate VAX code for the subtract instruction:

- Rules 3 and 4 form a loop. The potential loop: temp→operand→temp→operand... is broken by the matcher recognizing that the cost of the second match of temp does not cost less than the first match of temp.

- In the cost clause of rule 5, the cost is the sum of the leaves plus the cost of the subtract instruction. The action clause emits code to add the two operands and leave the result in a temporary location. The temporary is returned as a substitution for the subject tree.

- Rule 6 handles a special case where the left operand is already in a temporary and the constant is one. In this case, the temporary is directly decremented and returned as the new tree.

6 How to run `Twig`

Once a user has written the specification, `twig` is used to compile it into a subroutine. Figure 1 gives an overview of how `twig` does this. In Figure 1, rounded boxes indicate files and sharp cornered boxes are processing programs. The user creates the *twig* specification which is represented by the top box. The specification file must have the suffix "`.mt`". Let's say we have one called `arbor.mt`. To invoke the `twig` compiler we type:

twig [–**w**xx] arbor.mt

```
prologue        {
#include        "otherdefs.h"
union node {
                ...                             /* definition of other node types */
                union {
                                int operation;
                                NODEPTR left, right;
                                LABEL falselab, truelab;      /* LABEL will defined elsewhere */
                                ...                           /* other definitions relevant to test nodes */
                } test;         ...                           /* definition of other node types */
};
};
node            nAnd nOr nNot nLess;
label           lTest lExpr;
lTest:          nAnd(lTest,lTest)
                { TOPDOWN; }
                ={
                                $1$ →test.falselab = $$ →test.falselab;
                                $1$ →test.truelab = getlabel();
                                tDO($%1$);
                                printlab($1$ →test.truelab);
                                $2$ →test.falselab = $$ →test.falselab;
                                $2$ →test.truelab = $$ →test.truelab;
                                tDO($%2$);
                };
lTest:          nOr(lTest,lTest)
                { TOPDOWN; }
                ={
                                $1$ →test.falselab = getlabel();
                                $1$ →test.truelab = $$ →test.truelab;
                                tDO($%1$);
                                $2$ →test.truelab = $$ →test.truelab;
                                $2$ →test.falselab = $$ →test.falselab;
                                tDO($%2$);
                };
```

Figure 5: Short circuited Boolean Evaluation

```
lTest:      nNot(lTest)
            { TOPDOWN; }
            ={
                        $1$ →test.truelab = $$ →test.falselab;
                        $1$ →test.falselab = $$ →test.truelab;
                        tDO($%1$);
            };
lTest:      nLess(lExpr,lExpr)          /* not topdown anymore */
            ={ ... generate code for comparison ... };
```

Figure 5: (cont.) Short circuited Boolean Evaluation

The proprocessor reads `arbor.mt` and if there are no errors will create two files: `symbols.h` and `walker.c`. To build `walker.c`, `twig` uses a template file called `walker.`*xx* where *xx* is the argument of the optional –**w** flag. If the flag is omitted then *xx* defaults to "`c1`".

`Symbols.h` and `walker.c` can then compiled and linked with the other modules of the user's program. These other modules should provide the Tree and Cost ADTs described above. `Walker.h` is an auxilliary include file that is referenced by `walker.c`. A typical compile command is:

cc – **I***include_dir* **-c** walker.c

The –**I***include_dir* argument causes **cc** to look for walker.h in *include_dir*. The exact value of *include_dir* will depend on where the files are stored at your site. The tree matcher is initialized by calling `_matchinit` and matching can begin by calling `_match` from the user program.

7 Organization of the `Twig` Compiler

The preprocessor is written completely in C and YACC. It can be roughly broken up into four modules:

- Lexer — The lexer performs lexical analysis. The input is tokenized by this module and passed to the parser as required. Identifiers are looked up in the symbol table and space is reserved for them if they have never been encountered.

- Parser — This part is written in YACC and recognizes the *twig* language. When a language construct is recognized actions are invoked. Declarations, prologues and inserts will invoke symbol table operations when recognized. Rules are broken up and

```
prologue      {
/* otherdefs.h will have a type definition for NODEPTR */
#include      "otherdefs.h"
#define       TEMP_COST              5
#define       SUB_COST               30
#define       DEC_COST               10
NODEPTR       gettemp();
};
node          long constant sub;
label         operand temp;
operand:      long;                                 /* rule 1 */
operand:      constant;                             /* rule 2 */
operand:      temp;                                 /* rule 3 */
temp:         operand                               /* rule 4 */
              { cost = TEMP_COST+$%1$ →cost;}
              = {
                        NODEPTR t = gettemp();
                        emit("mov", $$, t, 0);
                        return(t);
              };
operand:      sub(operand,operand)                  /* rule 5 */
              {cost = $%1$ →cost + $%2$ →cost+SUB_COST;}
              ={
                        NODEPTR t = gettemp();
                        emit("sub", $1$, $2$, t, 0);
                        return(t);
              };
temp:         sub(temp,constant)                    /* rule 6 */
              {
                        if(value($2$)==1)
                                  cost = $%1$ →cost+DEC_COST;
                        else ABORT;
              }={
                        emit("dec", $1$, 0);
                        return($1$);
              };
```

Figure 6: Small Code Generation Example

stored in the symbol table too. The tree patterns in the rules are unflattened and passed to the machine builder.

- Machine Builder — This module takes tree patterns and incrementally builds an internal representation of the matching automaton. The builder takes this tree and then enumerates each of the possible paths from the root to the leaves. These paths are treated as strings and a string matching automaton is built as described in [1] and [7].

  Inside the builder the partially built machine is kept in a linked list form. Each machine state is a linked list and each node represents a state transition. When the tables for the machine are written out into `walker.c`, the linked list structure is translated into a table of sixteen bit integers. Each transistion is stored as two integers. The first is the integer corresponding to the symbol that causes the transistion and the second is the index of the next state in the table.

- Symbol Table — The central data structure in this symbol table is the hash table. Each entry in the hash table is a bucket — a linked list of symbol table entries. There is no rehashing. All colliding items are placed in the linked list. This is a simple and adequately efficient arrangement. Depending on the type of symbols, the entry may point to other data structures. Entries for *label_id*s point to lists of trees. *Node_id* entries record the integers that have been associates with them. Other entries may point to data structures holding a textual representation of C code that forms action and cost parts.

Here is a list of the files that form the `twig` compiler's source and their approximate function.

- `common.h` — This file contains data definitions for how trees are represented in the parser. It also contains type definitions for external functions.
- `code.h` — This file contains definitions for how code fragments are stored.
- `sym.h` — This file defines the major data structures in the symbol table.
- `machine.h` — This file defines the how the matching automaton is represented internally.
- `twig.y` — This is the parser.
- `sym.c` — This is the symbol table manager.
- `path.c` — The path string enumerator.
- `machine.c` — The machine builder.
- `lex.c` — The lexer.
- `tree.c`, `io.c`, and `code.c` — Miscellaneous routines for manipulating trees, performing I/O and manipulating code fragments.

8 Performance

So far, the only experience we have with *twig* is a VAX code generator written for the pcc2 compiler. The *twig* code generator is 25% faster than the original code generator. The maintainability and modifiability of the code generator has improved. For example adding the indexed addressing mode into the code generator took only a few hours. The target code quality is just as good as the original code generator.

The `twig` table generation algorithm is fast. For the VAX machine description which consist of 109 rules, the generation time was 4.2 seconds on a 780. The generation time could be increased by two orders of magnitude before other table driven system can compete with `twig` on this basis. Furthermore, the tables are small. The VAX description is only 7.5K and the text space for the walker is 30K. Again this is much smaller than those of other table driven systems.

Twig is currently a research tool. Several things can be done to improve *twig*'s performance.

- *Twig* does a lot of procedure calls. Every machine transition require at least one procedure call. Contrast this to YACC where a machine transition is done in line.

- *Twig* performs a very expensive closure operation with respect to unit rules. A unit rule is the analogue of a unit production and has the form:

  $label_1$: $label_2$;

  This closure is done at run time. It requires many procedure calls and manipulates complex data structures. We are looking at ways of doing this at table generation time or to hash the results of the closures so that redundant calculations can be avoided. The problem with doing them in the table generator is the possible explosion in its running time.

- Many of the cost parts for the rules are similar and hence some tests are performed and recalculated many times on a node. `Twig` should be clever enough to factor out some of these tests and do them only once. However, the information required for `twig` to do this is not easily available. It is hidden in the C code fragments.

- The current compact representation of the matching automaton require linear searching except at the start state where the large branching factor compelled us to use an array scheme. Using another representation would speed up the matcher. For example, implementing the VAX description as an array is estimated to use about 50K bytes but may provide performance improvement.

References

[1] Alfred V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliogrpahic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching. to be published.

[3] Alfred V. Aho and Stephen C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):458–501, July 1976.

[4] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison Wesley, April 1979.

[5] Mahadevan Ganapathi. *Retargetable Code Generator and Optimization using Attribute Grammars*. PhD thesis, University of Wisconsin – Madision, 1980.

[6] Robert Rettig Henry. *Graham-Glanville Code Generators*. PhD thesis, University of California at Berkeley, May 1984.

[7] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[8] Stephen C. Johnson. Yacc – yet another compiler-compiler. Comp. Sci. Tech. Rep. 32, AT&T Bell Laboratories, July 1975.