# Engineering a Simple, Efficient Code Generator Generator

CHRISTOPHER W. FRASER
AT&T Bell Laboratories
and
DAVID R. HANSON
Princeton University
and
TODD A. PROEBSTING
The University of Arizona

Many code generator generators use tree pattern matching and dynamic programming. This note describes a simple program that generates matchers that are fast, compact, and easy to understand. It is simpler than common alternatives: 200–700 lines of Icon or 950 lines of C versus 3000 lines of C for Twig and 5000 for `burg`. Its matchers run up to 25 times faster than Twig's. They are necessarily slower than `burg`'s BURS (bottom-up rewrite system) matchers but they are more flexible and still practical.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors — *code generation, compilers, translator writing systems and compiler generators*

General terms: Languages

Additional Key Words and Phrases: code generation, code generator generator, dynamic programming, tree pattern matching, Icon programming language

## 1. INTRODUCTION

Many code generator generators use tree pattern matching and dynamic programming (DP) [3, 4, 8]. They accept tree patterns and associated costs and semantic actions that, for example, allocate registers and emit object code. They produce tree matchers that make two passes over each subject tree. The first pass is bottom up and finds a set of patterns that cover the tree with minimum cost. The second pass executes the semantic actions associated with minimum-cost patterns at the nodes they matched. Code generator generators based on this model include BEG [7], Twig [2], and `burg` [13].

BEG matchers are hard-coded and mirror the tree patterns in the same way that recursive-descent parsers mirror their input grammars. They use

Authors' addresses: C. W. Fraser, AT&T Bell Laboratories, 600 Mountain Avenue 2C-464, Murray Hill, NJ 07974-0636; D. R. Hanson, Department of Computer Science, Princeton University, Princeton, NJ 08544; T. A. Proebsting, Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

dynamic programming at compile time to identify a minimum-cost cover.

Twig matchers use a table-driven variant of string matching [1, 15] that, in essence, identifies all possible matches at the same time. This algorithm is asymptotically better than trying each possible match one at a time, but overhead is higher. Like BEG matchers, Twig matchers use DP at compile time to identify a minimum-cost cover.

`burg` uses BURS (bottom-up rewrite system) theory [5, 6, 17, 18] to move the DP to compile-compile time. BURS table generation is more complicated, but BURS matchers generate optimal code in constant time per node. The main disadvantage of BURS is that costs must be constants; systems that delay DP until compile time permit costs to involve arbitrary computations.

This paper describes a program called `iburg` that reads a `burg` specification and writes a matcher that does DP at compile time. The matcher is hard coded, a technique that has proven effective with other types of code generators [9, 12]. `iburg` was built to test early versions of what evolved into `burg`'s specification language and interface, but it is useful in its own right because it is simpler and thus easier for novices to understand, because it allows dynamic cost computation, and because it admits a larger class of tree grammars [16]. `iburg` has been used with good results in a first course on compilers. `burg` and `iburg` have been used also to produce robust VAX, MIPS, and SPARC code generators for `lcc`, a retargetable compiler for ANSI C [11].

`iburg` and BEG produce similar matchers, but this note describes them in more detail than the standard BEG reference [7]. In particular, it describes several optimizations that paid off and two that did not, and it quantifies the strengths and weaknesses of such programs when compared with programs like Twig and `burg`.

## 2. SPECIFICATIONS

Figure 1 shows an extended BNF grammar for `burg` and `iburg` specifications. Grammar symbols are displayed in *slanted* type and terminal symbols are displayed in `typewriter` type. $\{X\}$ denotes zero or more instances of $X$, and $[X]$ denotes an optional $X$. Specifications consist of declarations, a `%%` separator, and rules. The declarations declare terminals — the operators in subject trees — and associate a unique, positive *external symbol number* with each one. Non-terminals are declared by their presence on the left side of rules. The `%start` declaration optionally declares a non-terminal as the start symbol. In Figure 1, *term* and *nonterm* denote identifiers that are terminals and non-terminals, respectively.

Rules define tree *patterns* in a fully parenthesized prefix form. Every non-terminal denotes a tree. Each operator has a fixed arity, which is inferred from the rules in which it is used. A *chain rule* is a rule whose pattern is another non-terminal. If no start symbol is declared, the non-terminal defined by the first rule is used.

Each rule has a unique, positive *external rule number*, which comes after the pattern and is preceded by a "`=`". As described below, external rule numbers are used to report the matching rule to a user-supplied semantic action routine. Rules end with an optional non-negative, integer cost; omitted costs default to zero.

```
grammar  →  { dcl } %% { rule }
dcl      →  %start nonterm
         |  %term { identifier = integer }
rule     →  nonterm : tree = integer [ cost ] ;
cost     →  ( integer )
tree     →  term ( tree , tree )
         |  term ( tree )
         |  term
         |  nonterm
```

Fig. 1. EBNF Grammar for `burg` and `iburg` Specifications.

```
1.  %term ADDI=309 ADDRLP=295 ASGNI=53
2.  %term CNSTI=21 CVCI=85 IOI=661 INDIRC=67
3.  %%
4.  stmt:  ASGNI(disp,reg) = 4 (1);
5.  stmt:  reg = 5;
6.  reg:   ADDI(reg,rc) = 6 (1);
7.  reg:   CVCI(INDIRC(disp)) = 7 (1);
8.  reg:   IOI = 8;
9.  reg:   disp = 9 (1);
10. disp:  ADDI(reg,con) = 10;
11. disp:  ADDRLP = 11;
12. rc:    con = 12;
13. rc:    reg = 13;
14. con:   CNSTI = 14;
15. con:   IOI = 15;
```

Fig. 2. Sample `burg` Specification.

Figure 2 shows a fragment of a `burg` specification for the VAX. This example uses upper-case for terminals and lower-case for non-terminals. Lines 1–2 declare the operators and their external symbol numbers, and lines 4–15 give the rules. The external rule numbers correspond to the line numbers to simplify interpreting subsequent figures. In practice, these numbers are usually generated by a preprocessor that accepts a richer form of specification (e.g., including YACC-style semantic actions), and emits a `burg` specification [13]. Only the rules on lines 4, 6, 7, and 9 have non-zero costs. The rules on lines 5, 9, 12, and 13 are chain rules.

The operators in Figure 2 are some of the operators in `lcc`'s intermediate language [10]. The operator names are formed by concatenating a generic operator name with a one-character type suffix like `C`, `I`, or `P`, which denote character, integer, and pointer operations, respectively. The operators used in Figure 2 denote integer addition (`ADDI`), forming the address of a local variable (`ADDRLP`), integer assignment (`ASGNI`), an integer constant (`CNSTI`), "widening" a character to an integer (`CVCI`), the integer `0` (`IOI`), and fetching a character (`INDIRC`). The rules show that `ADDI` and `ASGNI` are binary, `CVCI` and `INDIRC` are unary, and `ADDRLP`, `CNSTI`, and `IOI` are leaves.

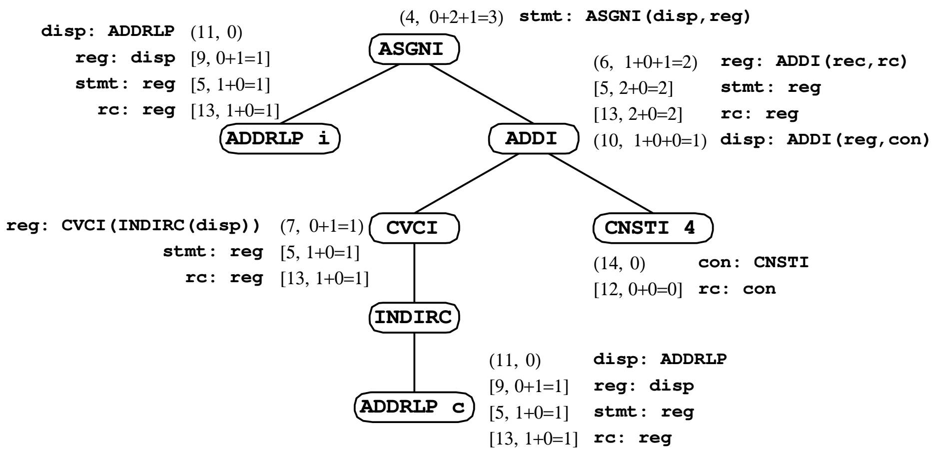


Fig. 3. Intermediate Language Tree for `i = c + 4`.

## 3. MATCHING

Both versions of `burg` generate functions that the client calls to *label* and *reduce* subject trees. The labeling function, `label (p)`, makes a bottom-up, left-to-right pass over the subject tree `p` computing the rules that cover the tree with the minimum cost, if there is such a cover. Each node is labeled with $(M, C)$ to indicate that "the pattern associated with external rule $M$ matches the node with cost $C$."

Figure 3 shows the intermediate language tree for the assignment expression in the C fragment

```
{ int i; char c; i = c + 4; }
```

The left child of the `ASGNI` node computes the address of `i`. The right child computes the address of `c`, fetches the character, widens it to an integer, and adds `4` to the widened value, which the `ASGNI` assigns to `i`.

The other annotations in Figure 3 show the results of labeling. $(M, C)$ denote labels from matches and $[M, C]$ denote labels from chain rules. The rule from Figure 2 denoted by each $M$ is also shown. Each $C$ sums the costs of the non-terminals on right-hand side and the cost of the relevant pattern or chain rule. For example, the pattern in line 11 of Figure 2 matches the node `ADDRLP i` with cost 0, so the node is labeled with $(11, 0)$. Since this pattern denotes a `disp`, the chain rule in line 9 applies with a cost of 0 for matching a `disp` plus 1 for the chain rule itself. Likewise, the chain rules in lines 5 and 13 apply because the chain rule in line 9 denotes a `reg`.

Patterns can specify subtrees beyond the immediate children. For example, the pattern in line 7 of Figure 2 refers to the grandchild of the `CVCI` node. No separate pattern matches the `INDIRC` node, but line 7's pattern covers that node. The cost is the cost of matching the `ADDRLP i` as a `disp`, which is rule 11, plus 1.

Nodes are annotated with $(M, C)$ only if $C$ is less than all previous matches for the non-terminal on the left-hand side of rule $M$. For example, the `ADDI` node matches the `disp` pattern in line 10 of Figure 2, which means it also matches all rules with `disp` alone on the right-hand side, namely line 9. By transitivity, it also matches the chain rules in lines 5 and 13. But all three of these chain rules yield cost 2, which isn't better than previous matches for those non-terminals.

Once labeled, a subject tree is reduced by traversing it from the top down and performing appropriate semantic actions, such as generating and emitting code. Reducers are supplied by clients, but `burg` generates functions that assist in these traversals, e.g., one function that returns $M$ and another that identifies subtrees for recursive visits. Reference [13] elaborates.

`burg` does all dynamic programming at compile-compile time and annotates each node with a single, integral *state* number, which encodes all of the information concerning matches and costs. `iburg` does the dynamic programming at compile time and annotates nodes with data equivalent to $(M, C)$. Its "state numbers" are really pointers to records that hold these data.

Both versions of `burg` generate an implementation of `label` that accesses node fields via client-supplied macros or functions and uses the non-recursive function `state` to identify matches:

```
int label(NODEPTR_TYPE p) {
   if (p) {
      int l = label(LEFT_CHILD(p));
      int r = label(RIGHT_CHILD(p));
      return STATE_LABEL(p) = state(OP_LABEL(p), l, r);
   } else
      return 0;
}
```

`NODEPTR_TYPE` is a `typedef` or macro that defines the data type of nodes, `OP_LABEL`, `LEFT_CHILD`, and `RIGHT_CHILD` are macros or functions that return, respectively, a node's external symbol number, its left child, and its right child. `STATE_LABEL` is a macro that accesses a node's state number field.

`state` accepts an external symbol number for a node and the state numbers for the node's left and right children. It returns the state number to assign to that node. For unary operators and leaves, it ignores the last one or two arguments, respectively.

## 4. IMPLEMENTATION

`iburg` generates a `state` function that uses a straightforward implementation of tree pattern matching [7]. It generates hard code instead of tables. Its "state numbers" are pointers to state records, which hold vectors of the $(M, C)$ values for successful matches. The state record for the specification in Figure 2 is

```
struct state {
   int op;
   struct state *left, *right;
   short cost[6];
   short rule[6];
};
```

iburg also generates integer codes for the non-terminals, which index the cost and rule vectors:

```
#define stmt_NT 1
#define disp_NT 2
#define rc_NT 3
#define reg_NT 4
#define con_NT 5
```

By convention, the start non-terminal has value 1.

State records are cleared when allocated, and external rule numbers are positive. Thus, a non-zero value for p->rule[$X$] indicates that p's node matched a rule that defines non-terminal $X$.

Figure 4 shows the implementation of state and gives the cases that are contributed by Figure 2's lines 6, 7, 10, and 11. state allocates and initializes a new state record and switches on the external symbol number to begin matching. Each non-leaf case is one or more if statements that test for a match by consulting the state records of descendants. The switch by itself does all necessary testing for leaves.

If a match succeeds, the resulting cost is computed and record is called with the pointer to the state record, the code for the matching non-terminal, the cost, and the matching external rule number:

```
void record(struct state *p, int nt, int cost, int eruleno) {
   if (cost < p->cost[nt]) {
      p->cost[nt] = cost;
      p->rule[nt] = eruleno;
   }
}
```

The match is recorded only if its cost is less that previous matches. The elements of the cost vector are initialized with 32767 to represent infinite cost, so the first match is always recorded.

The first call to record is for the match itself; the other calls are for chain rules. For example, the second if statement in the ADDI case tests whether p's node matches the pattern in line 10. If it does, the first call to record records that the node matches a disp. The chain rule in line 9 says that a node matching a disp also matches a reg with an additional cost of 1, which gives rise to the second record call. Likewise, the last two calls to record are due to the chain rules in lines 5 and 13, which say that a node matching a reg also matches a stmt and an rc, both with an additional cost of 0. In general, there is a call to record for the transitive closure of all chain rules that reach the non-terminal defined by the match.

## 5. IMPROVEMENTS

The generated matcher described in the previous section is practical for many code-generation applications, and the generator itself is easy to implement. Students have replicated the version that emits the code shown in Figure 4 in a couple of weeks. iburg implements, however, several simple improvements that make the generated matchers smaller and faster. Even with the

```
int state(int op, int left, int right) {
   int c; struct state *l = (struct state *)left,
                       *r = (struct state *)right, *p;
   p = malloc(sizeof *p);
   p->op = op; p->left = l; p->right = r;
   p->rule[1] = ... = 0; p->cost[1] = ... = 32767;
   switch (op) {
   case ADDI:
      if (l->rule[reg_NT] && r->rule[rc_NT) {
         c = l->cost[reg_NT] + r->cost[rc_NT] + 1;
         record(p, reg_NT, c, 6);
         record(p, rc_NT, c + 0, 13);
         record(p, stmt_NT, c + 0, 5);
      }
      if (l->rule[reg_NT] && r->rule[con_NT]) {
         c = l->cost[reg_NT] + r->cost[con_NT] + 0;
         record(p, disp_NT, c, 10);
         record(p, reg_NT, c + 1, 9);
         record(p, rc_NT, c + 1 + 0, 13);
         record(p, stmt_NT, c + 1 + 0, 5);
      }
      break;
   case ADDRLP:
      c = 0;
      record(p, disp_NT, c, 11);
      record(p, reg_NT, c + 1, 9);
      record(p, rc_NT, c + 1 + 0, 13);
      record(p, stmt_NT, c + 1 + 0, 5);
      break;
   case CVCI:
      if (l->op == INDIRC && l->left->rule[disp_NT]) {
         c = l->left->cost[disp_NT] + 1;
         record(p, reg_NT, c, 7);
         record(p, rc_NT, c + 0, 13);
         record(p, stmt_NT, c + 0, 5);
      }
      break;
   ...
   }
   return (int)p;
}
```

Fig. 4. Implementation of `state`.

improvements below, iburg takes only 642 lines of Icon [14].

The short elements of the rule vector can accommodate any external rule number, but many non-terminals are defined by only a few rules. For example, only lines 10 and 11 in Figure 2 define disp, so only two bits are needed to record one of the two positive values. Definitions can be mapped into a compact range of integers and stored in minimum space in state records as bit fields, e.g.,

```
struct state {
   int op;
   struct state *left, *right;
   short cost[6];
   struct {
      unsigned int stmt:2;
      unsigned int disp:2;
      unsigned int rc:2;
      unsigned int reg:3;
      unsigned int con:2;
   } rule;
};
```

External rule numbers for matches are retrieved by calling `rule` with a state number and a goal non-terminal [13]. `iburg` generates an implementation of `rule` that uses tables to map the integers in the compact representation to external rule numbers, e.g.,

```
short decode_disp[] = { 0, 10, 11 };
short decode_rc[]   = { 0, 12, 13 };
short decode_stmt[] = { 0,  4,  5 };
short decode_reg[]  = { 0,  6,  7, 8, 9 };
short decode_con[]  = { 0, 14, 15 };

int rule(int state, int goalnt) {
   struct state *p = (struct state *)state;
   switch (goalnt) {
   case disp_NT:  return decode_disp[p->rule.disp];
   case rc_NT:    return decode_rc[p->rule.rc];
   case stmt_NT:  return decode_stmt[p->rule.stmt];
   case reg_NT:   return decode_reg[p->rule.reg];
   case con_NT:   return decode_con[p->rule.con];
   }
}
```

Packed `rule` numbers cannot be subscripted, so `record` and the code that tests for a match must be changed. This scheme can save much space for large specifications. For example, the VAX specification has 47 non-terminals and the encoding scheme reduces the size of its `rule` vector from 96 to 16 bytes.

Packing rule numbers can also save time: it takes longer to read, write and decode packed rule numbers, but the smaller structure can be initialized much faster, with a single structure copy. The original VAX matcher initialized `rule` with 47 assignments; a structure copy would have been slower. With packed fields, 47 assignments would be slower yet, but a 16-byte structure copy beats the original 47 assignments by a margin that swamps the other costs of using packed fields.

Initialization costs can be reduced further still: all costs must be set, but only the `rule` field for the start symbol needs initialization. The `rule` fields are read in only two places: the `rule` function above, and the tests for a match. The `rule` function is called during a top-down tree traversal, which always begins with the start symbol as the goal non-terminal. If it finds the initializer's zero rule number, then the tree failed to match, and no more fields should be examined anyway. The match tests require no rule initialization at all. They read the `rule` fields of descendants, and if they read garbage, then the descendants failed to match, and their costs will be infinite, which will prevent recording a false match. With this improved initializer, packing rule numbers no longer saves time, but it still saves space, and the time cost is so small that it could not be measured.

`record` can also be improved. If the cost test in `record` fails, the tests in the calls to `record` that implement its chain rules must fail too, because costs increase monotonically. These calls can be avoided if the cost test fails. Inlining `record` accommodates both this improvement and packed `rule`s. For example, the second `if` statement in the `ADDI` case in Figure 4 becomes

```
if (l->rule.reg && r->rule.con) {
   c = l->cost[reg_NT] + r->cost[con_NT] + 0;
   if (c < p->cost[disp_NT]) {  /* disp: ADDI(reg,con) */
      p->cost[disp_NT] = c;
      p->rule.disp = 1;
      closure_disp(p, c);
   }
}
```

`p->rule.disp` is set to 1 because `decode_disp` above maps 1 to external rule 10.

This code also shows a more compact approach to handling chain rules. For each non-terminal $X$ that can be reached via chain rules, `iburg` generates `closure_`$X$, which records the chain rule match if its cost is better than previous matches and, if applicable, calls another closure function. For example, the closure function for `disp` is

```
void closure_disp(struct state *p, int c) {
   if (c + 1 < p->cost[reg_NT]) {  /* reg: disp */
      p->cost[reg_NT] = c + 1;
      p->rule.reg = 4;
      closure_reg(p, c + 1);
   }
}
```

The incoming cost, `c`, is the cost of matching the right-hand side of the chain rule. This cost plus the cost of the chain rule itself, e.g., 1 for line 9's `reg: disp`, is the cost of *this* application of the chain rule, and this sum is passed to the next closure function. `closure_reg` handles both chain rules for `reg` (lines 5 and 13):

```
void closure_reg(struct state *p, int c) {
   if (c + 0 < p->cost[rc_NT]) {  /* rc: reg */
      p->cost[rc_NT] = c + 0;
      p->rule.rc = 2;
   }
   if (c + 0 < p->cost[stmt_NT]) {  /* stmt: reg */
      p->cost[stmt_NT] = c + 0;
      p->rule.stmt = 2;
   }
}
```

The final improvement saves times for leaves, which abound in subject trees from code generators. Computing and encoding all of the state record data about matches at compile-compile time is complicated [18]. Leaves, however, always match and the contents of the state record is easily computed by simulating the effect of the assignments and closure function calls shown above. The state records for leaves can thus be allocated and initialized at compile-compile time, e.g., the `ADDRLP` case in Figure 4 becomes

```
case ADDRLP: {
   static struct state z = { 295, 0, 0,
      {  0,
         1, /* stmt: reg */
         0, /* disp: ADDRLP */
         1, /* rc: reg */
         1, /* reg: disp */
         32767,
      },{2, /* stmt: reg */
         2, /* disp: ADDRLP */
         2, /* rc: reg */
         4, /* reg: disp */
         0,
      }};
   return (int)&z;
   }
```

The first three values initialize the `op`, `left`, and `right` fields of the `state` structure. The two brace-enclosed initializers give the `cost` and `rule` values, respectively. The code at the beginning of `state` (see Figure 4) that allocates and initializes a state record is not needed for leaves, so it is protected by a test that excludes leaf `ops`.

The table below traces the addition of each improvement above. The first column shows the number of lines of Icon in `iburg` and helps quantify implementation cost. The second column shows the number of object bytes in the resulting matcher. The third column times `lcc` in a typical cross-compilation for the VAX on a MIPS processor. The fourth column shows the time spent in the `state` and `rule` routines. All times are in seconds. Specifications for RISC machines would show smaller improvements.

| `iburg` size | matcher size | `lcc` time | matcher time | version |
|---|---|---|---|---|
| 566 | 240140 | 2.5 | .69 | original untuned version |
| 580 | 56304 | 2.4 | .59 | inline `record`, add closure routines |
| 580 | 56120 | 2.4 | .59 | initialize only one element of `rule` |
| 616 | 58760 | 2.2 | .39 | precompute leaf states |
| 642 | 66040 | 2.2 | .39 | pack rule numbers |

Closure routines save so much space because they implement chain rules in one place rather than in multiple `record` runs. The initialization improvement could not be measured in this trial, but it is trivial to implement and it must save something. On the other hand, packed rule numbers must have cost some time, but it appears small, and it cuts the size of the state structure by almost half.

Two proposed improvements proved uneconomical. First, the closure routines were inlined; measurements showed that the matcher was faster, but it was larger than even the initial version above. Independently, the closure routines were recoded to avoid tail recursion, but no speedup was measured. The recoding replaced each closure routine with a case in a `switch` statement, and the switch bound check added unnecessary overhead, so it is possible that a compiler implementation of tail recursion could do better, though large speedups seem unlikely.

Much of the processing done by `iburg` is straightforward. For example, parsing the input and writing the output account for 181 and 159 lines, respectively, in the 642-line final version. By way of comparison, a new version of `iburg` written in C is 950 lines, and a `burg` processor is 5100 lines of C [18].

## 6. DISCUSSION

`iburg` was built to test early versions of what evolved into `burg`'s specification language and interface. Initial tests used Twig and a Twig preprocessor, but Twig produced incorrect matchers for large CISC grammars. The error proved hard to find, so Twig was abandoned and `iburg` written. The initial version was completed in two days and 200 lines of Icon. The final, student-proof version with full `burg`-compatible debugging support is 642 lines.

Matchers generated by `iburg` are slower than those from `burg`. The table below shows the times for compiling the C programs in the SPEC benchmarks [19] with two versions of `lcc`. These times are for running only the compiler proper; preprocessing, assembly, and linking time are not included.

| Benchmark | `iburg` | `burg` |
|---|---|---|
| `001.gcc` | 90.2 | 77.9 |
| `008.espresso` | 28.3 | 24.6 |
| `022.li` | 8.9 | 8.0 |
| `023.eqntott` | 5.6 | 4.9 |

The compilations were done on an IRIS 4D/220GTX with 32MB running IRIX 3.3.2, and the times are *elapsed* time in seconds and are the lowest elapsed times over several runs on a lightly loaded machine. All reported runs achieved at least 96% utilization (i.e., the ratio of times $(user + system) / elapsed \geq 0.96$).

The differences in compilation times are due entirely to differences in the performance of the two matchers. Profiling shows that the execution time of `iburg`'s `rule` is nearly identical to `burg`'s `rule`. On these inputs, `iburg`'s matcher accounts for 8.5–12.4% of the execution time while `burg`'s accounts for only 1.1–2.0% making it roughly 6–12 times faster.

Comparable figures for Twig are unavailable because it did not correctly process large grammars, but before work with Twig was abandoned, a few measurements were taken. Using a nearly complete VAX grammar, `lcc` compiled one 2000-line module in 20.71 seconds using a Twig matcher and 5.35 using a matcher from the initial `iburg`; it spent 15.64 seconds in the Twig matcher and 0.85 in the `iburg` matcher. Using a partial MIPS grammar, `lcc` compiled the module in 9.19 seconds using a Twig matcher and 4.54 using a matcher from the initial `iburg`; it spent 4.04 seconds in the Twig matcher and 0.16 in the `iburg` matcher. Both versions of `lcc` used a naive emitter that was slowed by complex grammars, which is why the VAX compiler was so much slower. The figures in this paragraph are useful for comparing Twig with `iburg`, but the naive emitter makes them useless for comparisons with anything else.

A disadvantage of BURS matchers is that the costs must be constant because the dynamic programming is done at compile-compile time. Costs in Twig specifications, however, can involve arbitrary computation and depend on context. For example, the pattern

```
ASGNI(disp,CNSTI)
```

specifies a clear instruction if the constant is 0. Twig's cost computations can inspect the subject tree and return a cost of, say, 1 if the constant is 0 and infinity otherwise.

BURS specifications can handle this kind of context sensitivity with additional operators that identify the special cases. For example, before calling `state`, `lcc`'s labeling pass changes `CNSTI` to `IOI` if the constant is 0. Thus,

```
ASGNI(disp,IOI)
```

specifies a clear instruction.

Most context-sensitive cases that arise in code generation, even for CISC machines, can be handled similarly, perhaps with a few additional rules. For example, recognizing and using the VAX's indexed addressing mode takes 12 rules in `lcc`'s specification. `iburg` could easily be extended so that predicates could be specified and tested during matching, much like BEG's conditions [7].

`iburg` can be useful during development. The generated `state` and `label` functions are easy to read and to debug. Indeed, they mirror their specification in the same way that the code for a recursive-descent parser mirrors its LL(1) grammar. This attribute has made `iburg` ideal for teaching. It has been used in a course that previously used Twig, but students prefer `iburg`. When students make their inevitable mistakes with table-driven matcher like Twig's or `burg`'s, only inscrutable numbers from the table are available from the debugger. When they make mistakes with `iburg`, each node explicitly records the matching rules and costs for each non-terminal, so users can easily compare the matcher's actual operation with their expectations.

## Acknowledgments

Section 2 borrows from Reference [13], parts of which were written by Robert Henry. The C version of `iburg` is available for anonymous `ftp` in the directory `pub` from `ftp.cs.princeton.edu`.

## REFERENCES


1. AHO, A. V., AND CORASICK, M. J. Efficient string matching: An aid to bibliographic search. *Communications of the ACM 18*, 6 (June 1975), 333–340.
2. AHO, A. V., GANAPATHI, M., AND TJIANG, S. W. K. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems 11*, 4 (Oct. 1989), 491–516.
3. AHO, A. V., AND JOHNSON, S. C. Optimal code generation for expression trees. *Journal of the ACM 23*, 3 (July 1976), 488–501.
4. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, MA, 1986.
5. BALACHANDRAN, A., DHAMDHERE, D. M., AND BISWAS, S. Efficient retargetable code generation using bottom-up tree pattern matching. *Journal of Computer Languages 15*, 3 (1990), 127–140.
6. CHASE, D. R. An improvement to bottom-up tree pattern matching. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Munich, Jan. 1987), pp. 168–177.
7. EMMELMANN, H., SCHRÖER, F.-W., AND LANDWEHR, R. BEG — a generator for efficient back ends. *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24*, 7 (July 1989), 227–237.
8. FERDINAND, C., SEIDL, H., AND WILHELM, R. Tree automata for code selection. In *Code Generation — Concepts, Tools, Techniques, Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany* (May 1991), R. Giegerich and S. L. Graham, Eds., Springer-Verlag, pp. 30–50.
9. FRASER, C. W. A language for writing code generators. *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24*, 7 (July 1989), 238–245.
10. FRASER, C. W., AND HANSON, D. R. A code generation interface for ANSI C. *Software—Practice & Experience 21*, 9 (Sept. 1991), 963–988.
11. FRASER, C. W., AND HANSON, D. R. A retargetable compiler for ANSI C. *SIGPLAN Notices 26*, 10 (Oct. 1991), 29–43.
12. FRASER, C. W., AND HENRY, R. R. Hard-coding bottom-up code generation tables to save time and space. *Software—Practice & Experience 21*, 1 (Jan. 1991), 1–12.
13. FRASER, C. W., HENRY, R. R., AND PROEBSTING, T. A. BURG—Fast optimal instruction selection and tree parsing. *SIGPLAN Notices 27*, 4 (Apr. 1992), 68–76.
14. GRISWOLD, R. E., AND GRISWOLD, M. T. *The Icon Programming Language*, second ed. Prentice Hall, Englewood Cliffs, NJ, 1990.
15. HOFFMAN, C. M., AND O'DONNELL, M. J. Pattern matching in trees. *Journal of the ACM 29*, 1 (Jan. 1982), 68–95.
16. PELEGRÍ-LLOPART, E. *Tree Transformation in Compiler Systems.* PhD thesis, University of California, Berkeley, Berkeley, CA, Dec. 1987.
17. PELEGRÍ-LLOPART, E., AND GRAHAM, S. L. Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (San Diego, CA, Jan. 1988), pp. 294–308.
18. PROEBSTING, T. A. Simple and efficient BURS table generation. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices 27*, 6 (June 1992), 331–340.

19. Standards Performance Evaluation Corp. *SPEC Benchmark Suite Release 1.0*, Oct. 1989.

Corrigendum. Section 5 notes that the cost tests make it unnecessary to initialize most `rule` fields. BEG [7] carried this observation one step further: Tests like the outer `if` statement in the improved `ADDI` case in Section 5 need not test `rule` fields at all; the cost tests suffice. Such `if` statements are necessary only if there are embedded terminals to test, like the `INDIRC` in the rule on line 7 of Figure 2. This improvement has been added to `iburg`. Trials could not quantify an improvement, but it probably saves something, and it is easier to read.