Burg — Fast Optimal Instruction Selection and Tree Parsing

Christopher W. Fraser
AT&T Bell Laboratories
600 Mountain Avenue 2C-464
Murray Hill, NJ 07974-0636
cwf@research.att.com

Robert R. Henry
Tera Computer Company
400 N. 34th St., Suite 300
Seattle, WA 98103-8600
rrh@tera.com

Todd A. Proebsting
Dept. of Computer Sciences
University of Wisconsin
Madison, WI 53706
todd@cs.wisc.edu

December 1991

1 Overview

Burg is a program that generates a fast tree parser using BURS (Bottom-Up Rewrite System) technology. It accepts a cost-augmented tree grammar and emits a C program that discovers in linear time an optimal parse of trees in the language described by the grammar. Burg has been used to construct fast optimal instruction selectors for use in code generation. Burg addresses many of the problems addressed by Twig [AGT89, App87], but it is somewhat less flexible and much faster. Burg is available via anonymous `ftp` from `kaese.cs.wisc.edu`. The compressed `shar` file `pub/burg.shar.Z` holds the complete distribution.

This document describes only that fraction of the BURS model that is required to use Burg. Readers interested in more detail might start with Reference [BDB90]. Other relevant documents include References [Kro75, HO82, HC86, Cha87, PLG88, PL87, BMW87, Hen89, FH91, Pro91].

2 Input

Burg accepts a tree grammar and emits a BURS tree parser. Figure 1 shows a sample grammar that implements a very simple instruction selector. Burg grammars are structurally similar to Yacc's. Comments follow C conventions. Text between "`%{`" and "`%}`" is called the *configuration section*; there may be several such segments. All are concatenated and copied verbatim into the head of the generated parser, which is called Burm. Text after the second "`%%`", if any, is also copied verbatim into Burm, at the end.

The configuration section configures Burm for the trees being parsed and the client's environment. This section must define `NODEPTR_TYPE` to be a visible typedef symbol for a pointer to a node in the subject tree. Burm invokes `OP_LABEL(p)`, `LEFT_CHILD(p)`, and `RIGHT_CHILD(p)` to read the operator and children from the node pointed to by `p`. It invokes `PANIC` when it detects an error. If the configuration section defines these operations as macros, they are implemented in-line; otherwise, they must be implemented as functions. The section on diagnostics elaborates on `PANIC`.

```
%{
#define NODEPTR_TYPE treepointer
#define OP_LABEL(p) ((p)->op)
#define LEFT_CHILD(p) ((p)->left)
#define RIGHT_CHILD(p) ((p)->right)
#define STATE_LABEL(p) ((p)->state_label)
#define PANIC printf
%}
%start reg
%term Assign=1 Constant=2 Fetch=3 Four=4 Mul=5 Plus=6
%%
con:  Constant                = 1 (0);
con:  Four                    = 2 (0);
addr: con                     = 3 (0);
addr: Plus(con,reg)           = 4 (0);
addr: Plus(con,Mul(Four,reg)) = 5 (0);
reg:  Fetch(addr)             = 6 (1);
reg:  Assign(addr,reg)        = 7 (1);
```

Figure 1: A Sample Tree Grammar

BURM computes and stores a single integral *state* in each node of the subject tree. The configuration section must define a macro `STATE_LABEL(p)` to access the state field of the node pointed to by `p`. A macro is required because BURG uses it as an lvalue. A C `short` is usually the right choice; typical code generation grammars require 100–1000 distinct state labels.

The tree grammar follows the configuration section. Figure 2 gives an EBNF grammar for BURG tree grammars. Comments, the text between "`%{`" and "`%}`", and the text after the optional second "`%%`" are treated lexically, so the figure omits them. In the EBNF grammar, quoted text must appear literally, `Nonterminal` and `Integer` are self-explanatory, and `Term` denotes an identifier previously declared as a terminal. $\{X\}$ denotes zero or more instances of X.

Text before the first "`%%`" declares the start symbol and the terminals or operators in subject trees. All terminals must be declared; each line of such declarations begins with `%term`. Each terminal has fixed arity, which BURG infers from the rules using that terminal. BURG restricts terminals to have at most two children. Each terminal is declared with a positive, unique, integral *external symbol number* after a "`=`". `OP_LABEL(p)` must return the valid external symbol number for `p`. Ideally, external symbol numbers form a dense enumeration. Non-terminals are not declared, but the start symbol may be declared with a line that begins with `%start`.

Text after the first "`%%`" declares the rules. A tree grammar is like a context-free grammar: it has rules, non-terminals, terminals, and a special start non-terminal. The right-hand side of a rule, called the *pattern*, is a tree. Tree patterns appear in prefix parenthesized form. Every non-terminal denotes a tree. A chain rule is a rule whose pattern is another non-terminal. If no start symbol is declared, BURG uses the non-terminal defined by the first rule. BURG needs a single start symbol; grammars for which it is natural to use multiple start symbols must be augmented with an artificial start symbol that derives, with zero cost, the grammar's natural start symbols. BURM will automatically select one that costs least for any given tree.

```
grammar:  {dcl} '%%' {rule}

dcl:      '%start' Nonterminal
dcl:      '%term' { Identifier '=' Integer }

rule:     Nonterminal ':' tree '=' Integer cost ';'
cost:     /* empty */
cost:     '(' Integer { ',' Integer } ')'

tree:     Term '(' tree ',' tree ')'
tree:     Term '(' tree ')'
tree:     Term
tree:     Nonterminal
```

Figure 2: EBNF Grammar for Tree Grammars for Burg

Burg accepts no embedded semantic actions like Yacc's, because no one format suited all intended applications. Instead, each rule has a positive, unique, integral *external rule number*, after the pattern and preceded by a "=". Ideally, external rule numbers form a dense enumeration. Burm uses these numbers to report the matching rule to a user-supplied routine, which must implement any desired semantic action; see below. Humans may select these integers by hand, but Burg is intended as a *server* for building BURS tree parsers. Thus some Burg clients will consume a richer description and translate it into Burg's simpler input.

Rules end with a vector of non-negative, integer costs, in parentheses and separated by commas. If the cost vector is omitted, then all elements are assumed to be zero. Burg retains only the first four elements of the list. The cost of a derivation is the sum of the costs for all rules applied in the derivation. Arithmetic on cost vectors treats each member of the vector independently. The tree parser finds the cheapest parse of the subject tree. It breaks ties arbitrarily. By default, Burg uses only the *principal cost* of each cost vector, which defaults to the first element, but options described below provide alternatives.

3 Output

Burm traverses the subject tree twice. The first pass or *labeller* runs bottom-up and left-to-right, visiting each node exactly once. Each node is labeled with a state, a single number that encodes all full and partial optimal pattern matches viable at that node. The second pass or *reducer* traverses the subject tree top-down. The reducer accepts a tree node's state label and a *goal* non-terminal — initially the root's state label and the start symbol — which combine to determine the rule to be applied at that node. By construction, the rule has the given goal non-terminal as its left-hand side. The rule's pattern identifies the subject subtrees and goal non-terminals for all recursive visits. Here, a "subtree" is not necessarily an immediate child of the current node. Patterns with interior operators cause the reducer to skip the corresponding subject nodes, so the reducer may proceed directly to grandchildren, great-grandchildren, and so on. On the other hand, chain rules cause the reducer to revisit the current subject node, with a new goal non-terminal, so x is also regarded as a subtree of x.

As the reducer visits (and possibly revisits) each node, user-supplied code implements semantic action side effects and controls the order in which subtrees are visited. The labeller is self-contained, but the reducer combines code from BURG with code from the user, so BURM does not stand alone.

The BURM that is generated by BURG provides primitives for labelling and reducing trees. These mechanisms are a compromise between expressibility, abstraction, simplicity, flexibility and efficiency. Clients may combine primitives into labellers and reducers that can traverse trees in arbitrary ways, and they may call semantic routines when and how they wish during traversal. Also, BURG generates a few higher level routines that implement common combinations of primitives, and it generates mechanisms that help debug the tree parse.

BURG generates the labeller as a function named `burm_label` with the signature

```
extern int burm_label(NODEPTR_TYPE p);
```

It labels the entire subject tree pointed to by `p` and returns the root's state label. State zero labels unmatched trees. The trees may be corrupt or merely inconsistent with the grammar.

The simpler `burm_state` is `burm_label` without the code to traverse the tree and to read and write its fields. It may be used to integrate labelling into user-supplied traversal code. A typical signature is

```
extern int burm_state(int op, int leftstate, int rightstate);
```

It accepts an external symbol number for a node and the labels for the node's left and right children. It returns the state label to assign to that node. For unary operators, the last argument is ignored; for leaves, the last two arguments are ignored. In general, BURG generates a `burm_state` that accepts the maximum number of child states required by the input grammar. For example, if the grammar includes no binary operators, then `burm_state` will have the signature

```
extern int burm_state(int op, int leftstate);
```

This feature is included to permit future expansion to operators with more than two children.

The user must write the reducer, but BURM writes code and data that help. Primary is

```
extern int burm_rule(int state, int goalnt);
```

which accepts a tree's state label and a goal non-terminal and returns the external rule number of a rule. The rule will have matched the tree and have the goal non-terminal on the left-hand side; `burm_rule` returns zero when the tree labelled with the given state did not match the goal non-terminal. For the initial, root-level call, `goalnt` must be one, and BURM exports an array that identifies the values for nested calls:

```
extern short *burm_nts[] = { ... };
```

is an array indexed by external rule numbers. Each element points to a zero-terminated vector of short integers, which encode the goal non-terminals for that rule's pattern, left-to-right. The user needs only these two externals to write a complete reducer, but a third external simplifies some applications:

```
extern NODEPTR_TYPE *burm_kids(NODEPTR_TYPE p, int eruleno, NODEPTR_TYPE kids[]);
```

accepts the address of a tree `p`, an external rule number, and an empty vector of pointers to trees. The procedure assumes that `p` matched the given rule, and it fills in the vector with the subtrees (in the sense described above) of `p` that must be reduced recursively. `kids` is returned. It is not zero-terminated.

The simple user code below labels and then fully reduces a subject tree; the reducer prints the tree cover. `burm_string` is defined below.

```
parse(NODEPTR_TYPE p) {
  burm_label(p);   /* label the tree */
  reduce(p, 1, 0);  /* and reduce it */
}

reduce(NODEPTR_TYPE p, int goalnt, int indent) {
  int eruleno = burm_rule(STATE_LABEL(p), goalnt);  /* matching rule number */
  short *nts = burm_nts[eruleno];            /* subtree goal non-terminals */
  NODEPTR_TYPE kids[10];                               /* subtree pointers */
  int i;

  for (i = 0; i < indent; i++)
    printf(".");                                     /* print indented ... */
  printf("%s\n", burm_string[eruleno]);                /* ... text of rule */
  burm_kids(p, eruleno, kids);             /* initialize subtree pointers */
  for (i = 0; nts[i]; i++)               /* traverse subtrees left-to-right */
    reduce(kids[i], nts[i], indent+1);        /* and print them recursively */
}
```

The reducer may recursively traverse subtrees in any order, and it may interleave arbitrary semantic actions with recursive traversals. Multiple reducers may be written, to implement multi-pass algorithms or independent single-pass algorithms.

For each non-terminal x, BURG emits a preprocessor directive to equate `burm_`x`_NT` with x's integral encoding. It also defines a macro `burm_`x`_rule(a)` that is equivalent to `burm_rule(a,`x`)`. For the grammar in Figure 1, BURG emits

```
#define burm_reg_NT 1
#define burm_con_NT 2
#define burm_addr_NT 3
#define burm_reg_rule(a) ...
#define burm_con_rule(a) ...
#define burm_addr_rule(a) ...
```

Such symbols are visible only to the code after the second "`%%`". If the symbols `burm_`x`_NT` are needed elsewhere, extract them from the BURM source.

The `-I` option directs BURG to emit an encoding of the input that may help the user produce diagnostics. The vectors

```
extern char *burm_opname[];
extern char burm_arity[];
```

hold the name and number of children, respectively, for each terminal. They are indexed by the terminal's external symbol number. The vectors

```
extern char *burm_string[];
extern short burm_cost[][4];
```

hold the text and cost vector for each rule. They are indexed by the external rule number. The zero-terminated vector

```
extern char *burm_ntname[];
```

is indexed by `burm_`x`_NT` and holds the name of non-terminal x. Finally, the procedures

```
extern int burm_op_label(NODEPTR_TYPE p);
extern int burm_state_label(NODEPTR_TYPE p);
extern NODEPTR_TYPE burm_child(NODEPTR_TYPE p, int index);
```

are callable versions of the configuration macros. `burm_child(p,0)` implements `LEFT_CHILD(p)`, and `burm_child(p,1)` implements `RIGHT_CHILD(p)`. A sample use is the grammar-independent expression `burm_opname[burm_op_label(p)]`, which yields the textual name for the operator in the tree node pointed to by `p`.

A complete tree parser can be assembled from just `burm_state`, `burm_rule`, and `burm_nts`, which use none of the configuration section except `PANIC`. The generated routines that use the rest of the configuration section are compiled only if the configuration section defines `STATE_LABEL`, so they can be omitted if the user prefers to hide the tree structure from BURM. This course may be wise if, say, the tree structure is defined in a large header file with symbols that might collide with BURM's.

BURM selects an optimal parse without dynamic programming at compile time [AJ76]. Instead, BURG does the dynamic programming at compile-compile time, as it builds BURM. Consequently, BURM parses quickly. Similar labellers have taken as few as 15 instructions per node, and reducers as few as 35 per node visited [FH91].

4 Debugging

BURM invokes `PANIC` when an error prevents it from proceeding. `PANIC` has the same signature as `printf`. It should pass its arguments to `printf` if diagnostics are desired and then either abort (say via `exit`) or recover (say via `longjmp`). If it returns, BURM aborts. Some errors are not caught.

BURG assumes a robust preprocessor, so it omits full consistency checking and error recovery. BURG constructs a set of states using a closure algorithm like that used in LR table construction. BURG considers all possible trees generated by the tree grammar and summarizes infinite sets of trees with finite sets. The summary records the cost of those trees but actually manipulates the differences in costs between viable alternatives using a dynamic programming algorithm. Reference [Hen89] elaborates.

Some grammars derive trees whose optimal parses depend on arbitrarily distant data. When this happens, BURG and the tree grammar *cost diverge*, and BURG attempts to build an infinite set of states; it first thrashes and ultimately exhausts memory and exits. For example, the tree grammar in Figure 3 diverges, since non-terminals `green_reg` and `red_reg` derive identical infinite trees with different costs. If the cost of rule 31 is changed to 1, then the grammar does not diverge.

Practical tree grammars describing instruction selection do not cost-diverge because infinite trees are derived from non-terminals that model temporary registers. Machines can move data between different types of registers for a small bounded cost, and the rules for these instructions prevent divergence. For example, if Figure 3 included rules to move data between red and green registers, the grammar would not diverge. If a bonafide machine grammar appears to make BURG loop, try a host with more memory. To apply BURG to problems other than instruction selection, be prepared to consult the literature on cost-divergence [PL87].

5 Running BURG

BURG reads a tree grammar and writes a BURM in C. BURM can be compiled by itself or included

```
%term Const=17 RedFetch=20 GreenFetch=21 Plus=22
%%
reg: GreenFetch(green_reg) = 10 (0);
reg: RedFetch(red_reg) = 11 (0);

green_reg: Const = 20 (0);
green_reg: Plus(green_reg,green_reg) = 21 (1);

red_reg: Const = 30 (0);
red_reg: Plus(red_reg,red_reg) = 31 (2);
```

Figure 3: A Diverging Tree Grammar

in another file. When suitably named with the `-p` option, disjoint instances of BURM should link together without name conflicts. The command:

`burg` [*arguments*] [*file*]

invokes BURG. If a *file* is named, BURG expects its grammar there; otherwise it reads the standard input. The options include:

`-c` N Abort if any relative cost exceeds N, which keeps BURG from looping on diverging grammars. Several references [PLG88, Hen89, BDB90, Pro91] explain relative costs.

`-d` Report a few statistics and flag unused rules and terminals.

`-o` *file* Write parser into *file*. Otherwise it writes to the standard output.

`-p` *prefix* Start exported names with *prefix*. The default is `burm`.

`-t` Generates smaller tables faster, but all goal non-terminals passed to `burm_rule` must come from an appropriate `burm_nts`. Using `burm_`x`_NT` instead may give unpredictable results.

`-I` Emit code for `burm_arity`, `burm_child`, `burm_cost`, `burm_ntname`, `burm_op_label`, `burm_opname`, `burm_state_label`, and `burm_string`.

`-O` N Change the principal cost to N. Elements of each cost vector are numbered from zero.

`-=` Compare costs lexicographically, using all costs in the given order. This option slows BURG and may produce a larger parser. Increases range from small to astronomical.

6 Acknowledgements

The first BURG was adapted by the second author from his CODEGEN package, which was developed at the University of Washington with partial support from NSF Grant CCR-88-01806. It was unbundled from CODEGEN with the support of Tera Computer. The current BURG was written by the third author with the support of NSF grant CCR-8908355. The interface, documentation, and testing involved all three authors.

Comments from a large group at the 1991 Dagstuhl Seminar on Code Generation improved BURG's interface. Robert Giegerich and Susan Graham organized the workshop, and the International Conference and Research Center for Computer Science, Schloss Dagstuhl, provided an ideal environment for such collaboration. Beta-testers included Helmut Emmelmann, Dave Hanson, John Hauser, Hugh Redelmeier, and Bill Waite.

References

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AJ76] Alfred V. Aho and Steven C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):458–501, July 1976.

[App87] Andrew W. Appel. Concise specification of locally optimal code generators. Technical report CS-TR-080-87, Princeton University, 1987.

[BDB90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[BMW87] Jürgen Börstler, Ulrich Mönche, and Reinhard Wilhelm. Table compression for tree automata. Technical Report Aachener Informatik-Berichte No. 87-12, RWTH Aachen, Fachgruppe Informatik, Aachen, Fed. Rep. of Germany, 1987.

[Cha87] David R. Chase. An improvement to bottom up tree pattern matching. *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, January 1987.

[FH91] Christopher W. Fraser and Robert R. Henry. Hard-coding bottom-up code generation tables to save time and space. *Software—Practice&Experience*, 21(1):1–12, January 1991.

[HC86] Philip J. Hatcher and Thomas W. Christopher. High-quality code generation via bottom-up tree pattern matching. *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, January 1986.

[Hen89] Robert R. Henry. Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher. Technical Report 89-02-04, University of Washington Computer Science Department, Seattle, WA, February 1989.

[HO82] Christoph Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[Kro75] H. H. Kron. *Tree Templates and Subtree Transformational Grammars.* PhD thesis, UC Santa Cruz, December 1975.

[PL87] Eduardo Pelegri-Llopart. *Tree Transformations in Compiler Systems.* PhD thesis, UC Berkeley, December 1987.

[PLG88] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, January 1988.

[Pro91] Todd A. Proebsting. Simple and efficient BURS table generation. Technical report, Department of Computer Sciences, University of Wisconsin, 1991.