Code Generation by Tree Walking

Amey Karkare
karkare@cse.iitk.ac.in

April 5, 2019

Code Generation by Tree Walking

- Pushes dynamic programming to a pre-processing stage prior to code-generation time.

Code Generation by Tree Walking

- Pushes dynamic programming to a pre-processing stage prior to code-generation time.
- Simplifies dynamic programming effort by assuming unbounded number of registers.

Code Generation by Tree Walking

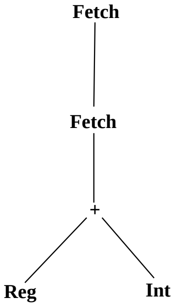
- Pushes dynamic programming to a pre-processing stage prior to code-generation time.
- Simplifies dynamic programming effort by assuming unbounded number of registers.
- Only cases taken into account are different patterns matching a node.

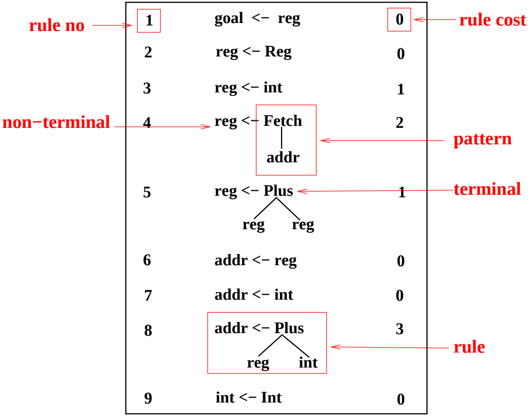
Code Generation by Tree Walking

- Pushes dynamic programming to a pre-processing stage prior to code-generation time.
- Simplifies dynamic programming effort by assuming unbounded number of registers.
- Only cases taken into account are different patterns matching a node.
- Say something about normalising instructions

Code Generation by Tree Walking – Example

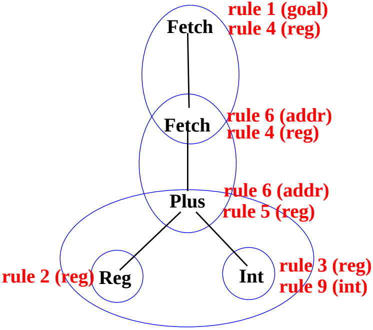
- An example expression tree and an example machine:



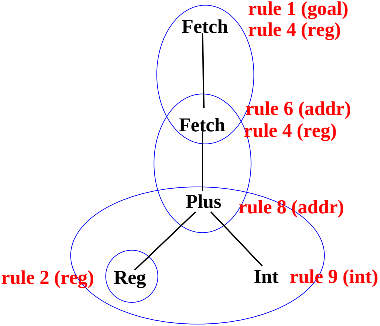


Code Generation by Tree Walking – Example

- The tree can be covered in more than one ways



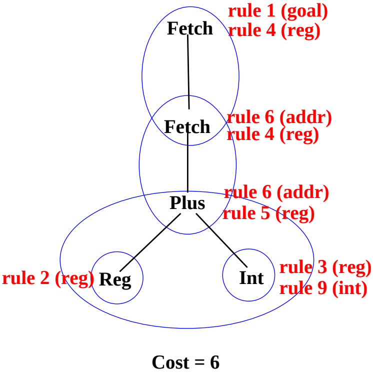
Cost = 6

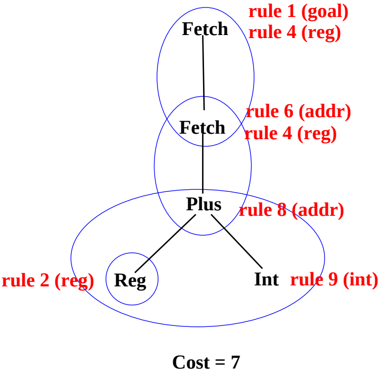


Cost = 7

Code Generation by Tree Walking – Example

- The tree can be covered in more than one ways

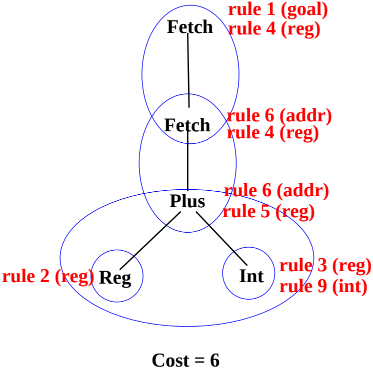


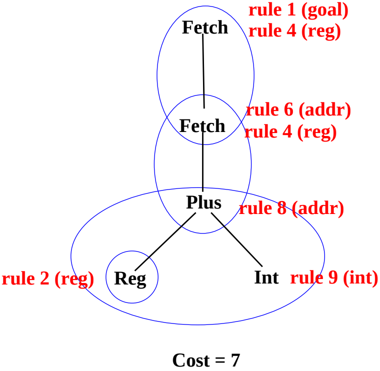


- We are finally interested in the least cost tree.

Code Generation by Tree Walking – Example

- The tree can be covered in more than one ways





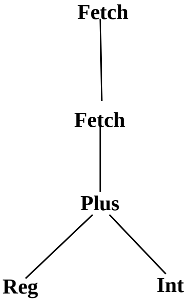
- We are finally interested in the least cost tree.
- We also want to do some pre-processing before we get any tree,

Code Generation by Tree Walking – Example

- How is this done? Given a tree,

Code Generation by Tree Walking – Example

- How is this done? Given a tree,
  - traverse the tree bottom up. With the help of a *transition table*, annotate each node of the tree with a state.



state 5:
reg <− Fetch(addr), 6
goal <− reg, 6
addr <− reg, 6

state 4:
reg <− Fetch(addr), 4
goal <− reg, 4
addr <− reg, 4

state 3:
reg <− Plus(reg, reg), 2
goal <− reg, 2
addr <− reg, 2

state 2:
reg <− int, 1
goal <− reg, 1
addr <− reg, 1
int <− Int, 0

state 1:
reg <− Reg, 0
goal <− reg, 0
addr <− reg, 0

Code Generation by Tree Walking

- *State:* Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.

Code Generation by Tree Walking

- *State:* Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.
- *Transition table:* Gives

Code Generation by Tree Walking

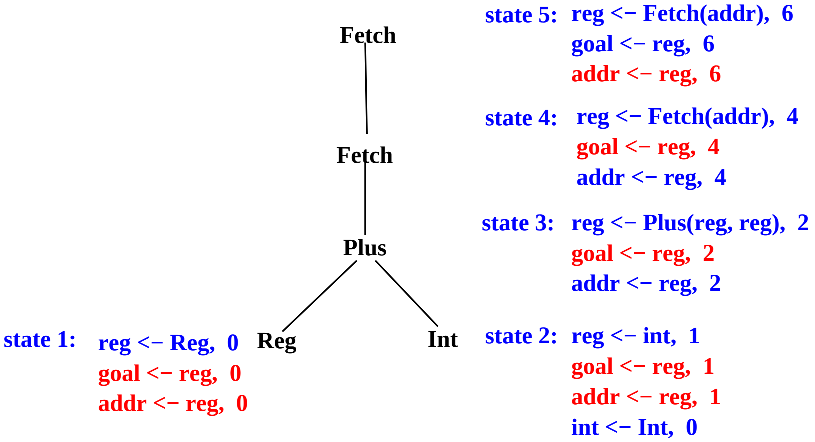
- *State:* Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.
- *Transition table:* Gives
  - state corresponding to leaf nodes (0-ary terminals).

Code Generation by Tree Walking

- *State:* Gives the minimum cost of evaluating a node in the expression tree to different non-terminals.
- *Transition table:* Gives
  - state corresponding to leaf nodes (0-ary terminals).
  - given the states of children, gives state of interior nodes (n-ary terminals).

Code Generation by Tree Walking – Example

- A second top-down pass determines the instructions to be used at each node assuming that the root is to be evaluated in goal.



Precomputing the Transition Table

- For 0-ary terminals

Precomputing the Transition Table

- For 0-ary terminals
  - Find least cost *covering rules*. A covering rule can cover the terminal with its pattern.

Precomputing the Transition Table

- For 0-ary terminals
  - Find least cost *covering rules*. A covering rule can cover the terminal with its pattern.
  - Find least cost *chain rules*. A chain rule is of the form *nonterminal* $\leftarrow$ *nonterminal*.

Int

```
goal <- reg, 1      -- chain rule
reg <- int, 1       -- chain rule
int <- Int, 0       -- covering rule
addr <- reg, 1      -- chain rule
```

Precomputing the Transition Table

- For 0-ary terminals
  - Find least cost *covering rules*. A covering rule can cover the terminal with its pattern.
  - Find least cost *chain rules*. A chain rule is of the form *nonterminal* $\leftarrow$ *nonterminal*.

Int

```
goal <- reg, 1      -- chain rule
reg <- int, 1       -- chain rule
int <- Int, 0       -- covering rule
addr <- reg, 1      -- chain rule
```

- Cost of reducing Int to goal is

  cost of reducing Int to int (0) $+$

  cost of reducing int to reg (1) $+$

  cost of reducing reg to goal (0) $+$

Precomputing the Transition Table

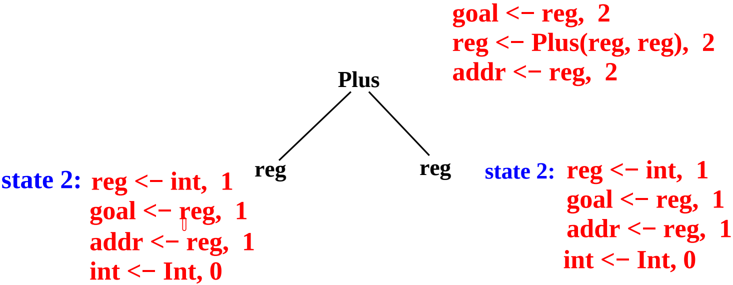
- n-ary terminals

Precomputing the Transition Table

- n-ary terminals
  - If both children of Plus are in state 2, in which state would Plus be?

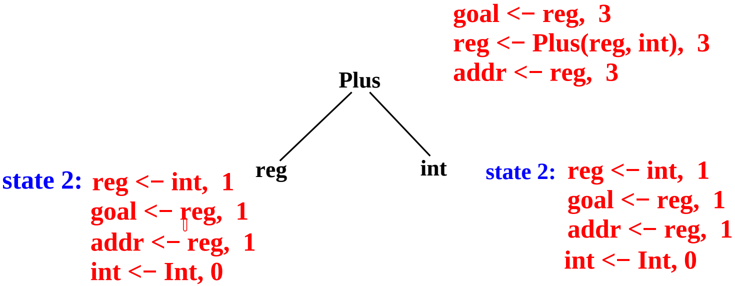
Precomputing the Transition Table

- n-ary terminals
  - If both children of Plus are in state 2, in which state would Plus be?
- The rule reg $\leftarrow$ Plus(reg, reg) gives



Precomputing the Transition Table

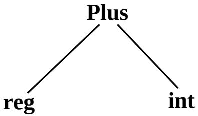
- The rule reg $\leftarrow$ Plus(reg, int) gives



Precomputing the Transition Table

- The rule reg $\leftarrow$ Plus(reg, int) gives

goal <− reg, 3
reg <− Plus(reg, int), 3
addr <− reg, 3



state 2: **reg <− int, 1**
goal <− reg, 1
addr <− reg, 1
int <− Int, 0

state 2: **reg <− int, 1**
goal <− reg, 1
addr <− reg, 1
int <− Int, 0

- Conclusion: If the leaves of Plus are both in state 2, then Plus will be in

state 6: **goal <− reg, 2**
reg <− Plus(reg, reg), 2
addr <− reg, 2

Precomputing the Transition Table

- Similarly, we should also find the transitions for Plus on pairs (state1, state1), (state1, state2), (state2, state6)

Precomputing the Transition Table

- Similarly, we should also find the transitions for Plus on pairs (state1, state1), (state1, state2), (state2, state6)
- ...

Precomputing the Transition Table

- Similarly, we should also find the transitions for Plus on pairs (state1, state1), (state1, state2), (state2, state6)
- . . .
- Will this process always terminate?

Precomputing the Transition Table

- Consider computation of the state at Fetch, with reg in the state shown.

Fetch

reg

reg <− Plus(reg, reg), 0
goal <− reg, 0
addr <− reg, 0

Precomputing the Transition Table

- Consider computation of the state at Fetch, with reg in the state shown.

Fetch

reg

reg <− Plus(reg, reg), 0
goal <− reg, 0
addr <− reg, 0

- Successive computation of the states for Fetch yield:

Fetch

reg <− Fetch(addr), 2
goal <− reg, 2
addr <− reg, 2

reg

reg <− Plus(reg, reg), 0
goal <− reg, 0
addr <− reg, 0

Fetch

reg <− Fetch(addr), 4
goal <− reg, 4
addr <− reg, 4

reg

reg <− Plus(reg, reg), 2
goal <− reg, 2
addr <− reg, 2

Relativization of states

- The solution is to relativize the costs in a state with respect to the item with the cheapest cost

Relativization of states

- The solution is to relativize the costs in a state with respect to the item with the cheapest cost
- After relativization, the state on the left changes to the state on the right:

reg <− Plus(reg, reg), 2	**reg <− Plus(reg, reg), 0**
goal <− reg, 2	**goal <− reg, 0**
addr <− reg, 2	**addr <− reg, 0**

Relativization of states

- The solution is to relativize the costs in a state with respect to the item with the cheapest cost
- After relativization, the state on the left changes to the state on the right:

```
reg <− Plus(reg, reg),  2
goal <− reg,  2
addr <− reg,  2
```

```
reg <− Plus(reg, reg),  0
goal <− reg,  0
addr <− reg,  0
```

  - Does this make the resulting transition table different? Obviously not.

Relativization of states

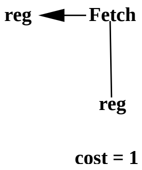
- The solution is to relativize the costs in a state with respect to the item with the cheapest cost
- After relativization, the state on the left changes to the state on the right:

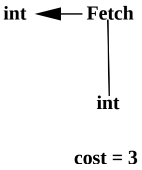
reg <− Plus(reg, reg), 2	**reg <− Plus(reg, reg), 0**
goal <− reg, 2	**goal <− reg, 0**
addr <− reg, 2	**addr <− reg, 0**

  - Does this make the resulting transition table different? Obviously not.
  - Does this necessarily lead to a finite number of states?

Relativization of states

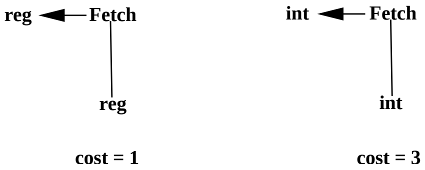
- Consider a machine with only these two instructions involving Fetch.





Relativization of states

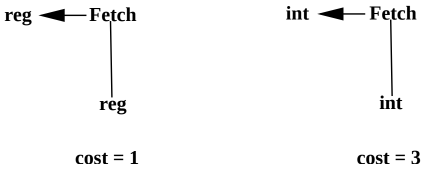
- Consider a machine with only these two instructions involving Fetch.



- Consider a state in which the reg $\leftarrow \ldots$ item is 2 cheaper than the int $\leftarrow \ldots$ item.

Relativization of states

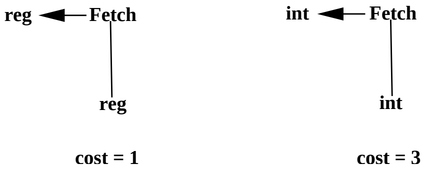
- Consider a machine with only these two instructions involving Fetch.



- Consider a state in which the reg $\leftarrow \dots$ item is 2 cheaper than the int $\leftarrow \dots$ item.
- Transits to a state in which the reg $\leftarrow \dots$ item is 4 cheaper than the int $\leftarrow \dots$ item.

Relativization of states

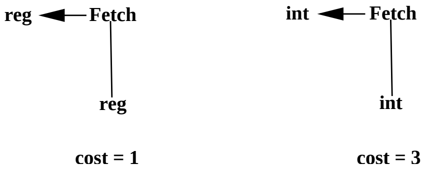
- Consider a machine with only these two instructions involving Fetch.



- Consider a state in which the reg $\leftarrow \ldots$ item is 2 cheaper than the int $\leftarrow \ldots$ item.
- Transits to a state in which the reg $\leftarrow \ldots$ item is 4 cheaper than the int $\leftarrow \ldots$ item.
- $\ldots$

Relativization of states

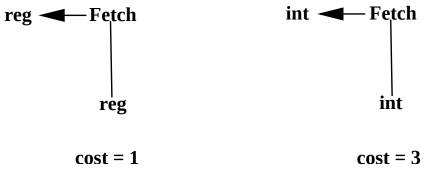
- Consider a machine with only these two instructions involving Fetch.



- Consider a state in which the reg $\leftarrow \ldots$ item is 2 cheaper than the int $\leftarrow \ldots$ item.
- Transits to a state in which the reg $\leftarrow \ldots$ item is 4 cheaper than the int $\leftarrow \ldots$ item.
- ...
- Practical solution: If cost difference between any pair of terminals is greater than a threshold, instruction set is rejected.

Relativization of states

- Consider a machine with only these two instructions involving Fetch.



- Consider a state in which the reg $\leftarrow \ldots$ item is 2 cheaper than the int $\leftarrow \ldots$ item.
- Transits to a state in which the reg $\leftarrow \ldots$ item is 4 cheaper than the int $\leftarrow \ldots$ item.
- ...
- Practical solution: If cost difference between any pair of terminals is greater than a threshold, instruction set is rejected.
- Typical instruction sets do not lead to divergence.

Relativization of states

- Naively generated transition tables are very large.

Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.

Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situtaions

Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situtaions
- Two major optimizations

Relativization of states

- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situtaions

- Two major optimizations
  - State reduction by projecting out irrelevant items

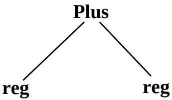
Relativization of states

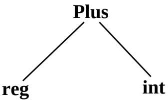
- Naively generated transition tables are very large.
- Typical CISC machine (1995 vintage) will generate 1000 states.
- Two states can be merged if the difference is not important in all possible situtaions

- Two major optimizations
  - State reduction by projecting out irrelevant items
  - State reduction by triangle trimming.

State Reduction by Projecting out Irrelevant Items

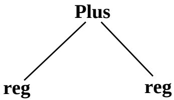
- Consider a machine in which the only instructions involving Plus are:

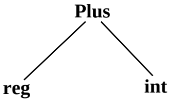




State Reduction by Projecting out Irrelevant Items

- Consider a machine in which the only instructions involving Plus are:





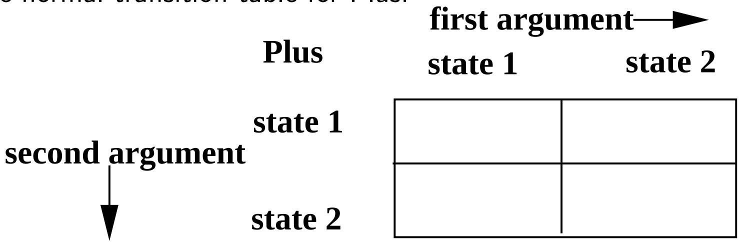
- Also assume that there are two states:

state 1:
goal <− reg, 0
reg <− Reg, 0
addr <− reg, 0

state 2:
goal <− reg, 1
reg <− int, 1
addr <− reg, 1
int <− Int, 0

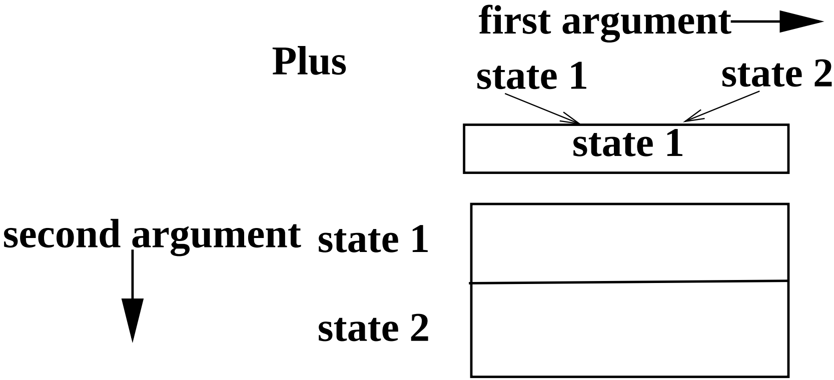
State Reduction by Projecting out Irrelevant Items

- The normal transition table for Plus:



State Reduction by Projecting out Irrelevant Items

- Since the first argument of Plus is a reg, we can project the int $\leftarrow \ldots$ item out of both the states. The resulting transition table for Plus is:

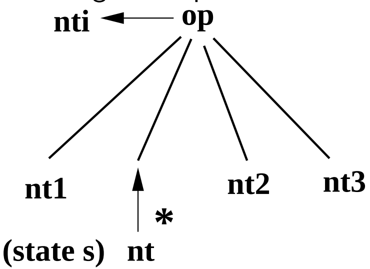


State Reduction by Triangle Trimming

- Assume that a state s has two items nt $\leftarrow \ldots$ and nt' $\leftarrow \ldots$. Under what conditions can we say that nt $\leftarrow \ldots$ is subsumed by nt' $\leftarrow \ldots$ and thus can be removed from s.

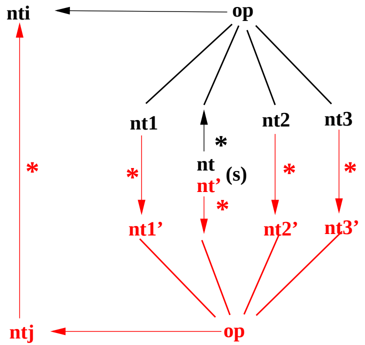
State Reduction by Triangle Trimming

- Assume that a state s has two items nt $\leftarrow \ldots$ and nt' $\leftarrow \ldots$. Under what conditions can we say that nt $\leftarrow \ldots$ is subsumed by nt' $\leftarrow \ldots$ and thus can be removed from s.
- Assume that the state has been used in the context of the operator op at the argument position shown



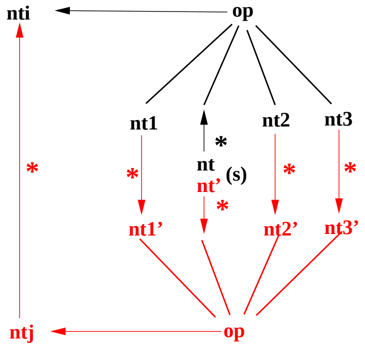
State Reduction by Triangle Trimming

- The general situation under which nt $\leftarrow \ldots$ is subsumed by nt' $\leftarrow \ldots$ is:



State Reduction by Triangle Trimming

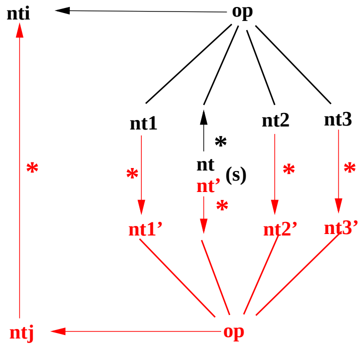
- The general situation under which nt $\leftarrow \ldots$ is subsumed by nt' $\leftarrow \ldots$ is:



- The cost of the rule nti $\leftarrow \ldots$ and the black chain reductions should be less than the rule ntj $\leftarrow \ldots$ and the red chain reductions.

State Reduction by Triangle Trimming

- The general situation under which nt $\leftarrow \ldots$ is subsumed by nt' $\leftarrow \ldots$ is:



- The cost of the rule nti $\leftarrow \ldots$ and the black chain reductions should be less than the rule ntj $\leftarrow \ldots$ and the red chain reductions.
- Further this should be true in all contexts in which s can be used.

BURG – A Code Generation Tool

- **B**ottom **U**p **R**ewriting based code **G**enerator

BURG – A Code Generation Tool

- **B**ottom **U**p **R**ewriting based code **G**enerator
- Sample BURG input.

BURG's name for node type

user's name for node type

macros to traverse tree

```
%{
  #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)
%}
```

terminals

```
  %start goal
  %term Assign=1 Constant=2 Fetch=3 Four=4
  %term Mul=5 Plus=6
```

rule number

cost

non−terminals

rule

pattern

```
  %%
  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);
  goal: reg                       = 8 (0);
```

BURG – A Code Generation Tool

- Two traversals over the subject tree.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function

      `burg_label(NODEPTR_TYPE p)`.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function

      `burg_label(NODEPTR_TYPE p)`.
    - Labels the subject tree with states (represented by integers).

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function

      `burg_label(NODEPTR_TYPE p)`.
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function

      `burg_label(NODEPTR_TYPE p).`
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function

      `reduce(NODEPTR_TYPE p, int goalInt)`

      written by user around BURG generated functions.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function

      `burg_label(NODEPTR_TYPE p)`.
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function

      `reduce(NODEPTR_TYPE p, int goalInt)`

      written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function

      `burg_label(NODEPTR_TYPE p)`.
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function

      `reduce(NODEPTR_TYPE p, int goalInt)`

      written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
    - At each node selects a rule for evaluating the node.

BURG – A Code Generation Tool

- Two traversals over the subject tree.
  - Labeling traversal.
    - Done entirely by generated function
      `burg_label(NODEPTR_TYPE p)`.
    - Labels the subject tree with states (represented by integers).
  - Rule selection traversal.
    - Done by a wrapper function
      `reduce(NODEPTR_TYPE p, int goalInt)`
      written by user around BURG generated functions.
    - Starts with the root of the subject tree and the non-terminal goal.
    - At each node selects a rule for evaluating the node.
    - Passes control back to user function with an integer identifying the rule. Actions corresponding to the rule to be managed by the user.

BURG – A Code Generation Tool

- Here is an outline of a code-generator produced

  with the help of BURG. Constructs in red are BURG generated.

```
parse(NODEPTR_TYPE p) {
  burg_label(p)   /* label the tree */
  reduce(p, 1)   /* and reduce it, goal = 1*/
}

reduce(NODEPTR_TYPE p, int goalint) {
  int ruleno = burg_rule(STATE_LABEL(p), goalint);
  short *nts = burg_nts[ruleno] ;
  NODEPTR_TYPE kids[10];
  int i;
  /* ... do something with this node... */

  /* process the children of this node  */
  burg_kids(p, ruleno, kids);
  for (i = 0; nts[i]; i++)
    reduce(kids[i], nts[i]);
}
```