Code Generation: Sethi Ullman Algorithm

Amey Karkare

karkare@cse.iitk.ac.in

March 28, 2019

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and
  - binary operations involving either a register and a memory, or two registers.

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and
  - binary operations involving either a register and a memory, or two registers.
- Does not use algebraic properties of operators. If $a * b$ has to be evaluated using $r_1 \leftarrow r_1 * r_2$, then a and b have to be necessarily loaded in r_1 and r_2 respectively.

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and
  - binary operations involving either a register and a memory, or two registers.
- Does not use algebraic properties of operators. If $a * b$ has to be evaluated using $r_1 \leftarrow r_1 * r_2$, then a and b have to be necessarily loaded in r_1 and r_2 respectively.
- Extensions to take into account algebraic properties of operators.

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and
  - binary operations involving either a register and a memory, or two registers.
- Does not use algebraic properties of operators. If $a * b$ has to be evaluated using $r_1 \leftarrow r_1 * r_2$, then a and b have to be necessarily loaded in r_1 and r_2 respectively.
- Extensions to take into account algebraic properties of operators.
- Generates optimal code – i.e. code with least number of instructions. There may be other notions of optimality.

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and
  - binary operations involving either a register and a memory, or two registers.
- Does not use algebraic properties of operators. If $a * b$ has to be evaluated using $r_1 \leftarrow r_1 * r_2$, then a and b have to be necessarily loaded in r_1 and r_2 respectively.
- Extensions to take into account algebraic properties of operators.
- Generates optimal code – i.e. code with least number of instructions. There may be other notions of optimality.
- Complexity is linear in the size of the expression tree. Reasonably efficient.

Expression Trees

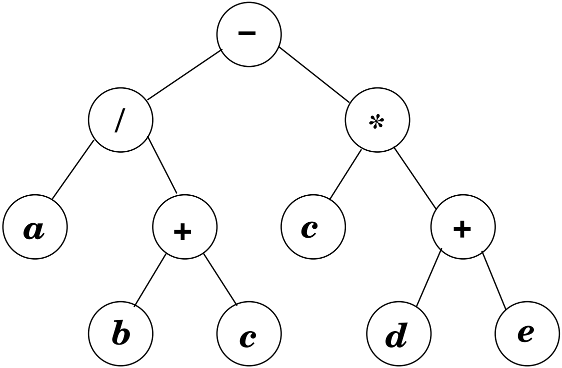
- Here is the expression $a/(b+c) - c * (d+e)$ represented as a tree:

Expression Trees

- Here is the expression $a/(b+c) - c * (d+e)$ represented as a tree:

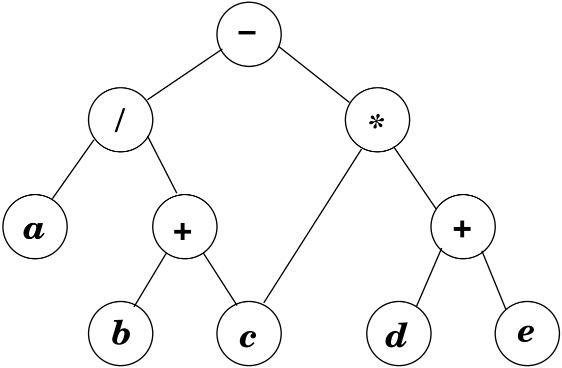
Expression Trees

- Here is the expression $a/(b+c) - c*(d+e)$ represented as a tree:



Expression Trees

- We have not identified common sub-expressions; else we would have a directed acyclic graph (DAG):



Expression Trees

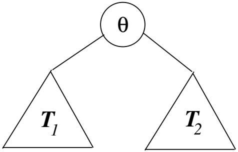
- Let Σ be a countable set of variable names, and Θ be a finite set of binary operators. Then,

Expression Trees

- Let Σ be a countable set of variable names, and Θ be a finite set of binary operators. Then,
  1. A single vertex labeled by a name from Σ is an expression tree.

Expression Trees

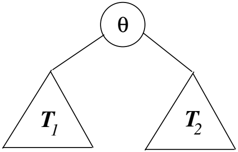
- Let Σ be a countable set of variable names, and Θ be a finite set of binary operators. Then,
  1. A single vertex labeled by a name from Σ is an expression tree.
  2. If T_1 and T_2 are expression trees and θ is a operator in Θ,

     then 

     is an expression tree.

Expression Trees

- Let Σ be a countable set of variable names, and Θ be a finite set of binary operators. Then,
  1. A single vertex labeled by a name from Σ is an expression tree.
  2. If T_1 and T_2 are expression trees and θ is a operator in Θ,

     then  is an expression tree.

- In this example

  $\Sigma = \{a, b, c, d, e, \ldots\}$, and $\Theta = \{+, -, *, /, \ldots\}$

Target Machine Model

- We assume a machine with finite set of registers r_0, r_1, ..., r_k, countable set of memory locations, and instructions of the form:

Target Machine Model

- We assume a machine with finite set of registers r_0, r_1, ..., r_k, countable set of memory locations, and instructions of the form:
  1. $m \leftarrow r$ (store instruction)

Target Machine Model

- We assume a machine with finite set of registers $r_0, r_1, \ldots, r_k$, countable set of memory locations, and instructions of the form:
  1. $m \leftarrow r$ (store instruction)
  2. $r \leftarrow m$ (load instruction)

Target Machine Model

- We assume a machine with finite set of registers r_0, r_1, ..., r_k, countable set of memory locations, and instructions of the form:
  1. $m \leftarrow r$ (store instruction)
  2. $r \leftarrow m$ (load instruction)
  3. $r \leftarrow r \; op \; m$ (the result of $r \; op \; m$ is stored in r)

Target Machine Model

- We assume a machine with finite set of registers r_0, r_1, ..., r_k, countable set of memory locations, and instructions of the form:
  1. $m \leftarrow r$ (store instruction)
  2. $r \leftarrow m$ (load instruction)
  3. $r \leftarrow r \; op \; m$ (the result of $r \; op \; m$ is stored in r)
  4. $r_2 \leftarrow r_2 \; op \; r_1$ (the result of $r_2 \; op \; r_1$ is stored in r_2)

Target Machine Model

- We assume a machine with finite set of registers $r_0, r_1, \ldots, r_k$, countable set of memory locations, and instructions of the form:
  1. $m \leftarrow r$ (store instruction)
  2. $r \leftarrow m$ (load instruction)
  3. $r \leftarrow r\ op\ m$ (the result of $r\ op\ m$ is stored in r)
  4. $r_2 \leftarrow r_2\ op\ r_1$ (the result of $r_2\ op\ r_1$ is stored in r_2)
- Note:

Target Machine Model

- We assume a machine with finite set of registers r_0, r_1, ..., r_k, countable set of memory locations, and instructions of the form:
  1. $m \leftarrow r$ (store instruction)
  2. $r \leftarrow m$ (load instruction)
  3. $r \leftarrow r \; op \; m$ (the result of $r \; op \; m$ is stored in r)
  4. $r_2 \leftarrow r_2 \; op \; r_1$ (the result of $r_2 \; op \; r_1$ is stored in r_2)
- Note:
  1. In instruction 3, the memory location is the right operand.

Target Machine Model

- We assume a machine with finite set of registers r_0, r_1, ..., r_k, countable set of memory locations, and instructions of the form:
  1. $m \leftarrow r$ (store instruction)
  2. $r \leftarrow m$ (load instruction)
  3. $r \leftarrow r \; op \; m$ (the result of $r \; op \; m$ is stored in r)
  4. $r_2 \leftarrow r_2 \; op \; r_1$ (the result of $r_2 \; op \; r_1$ is stored in r_2)

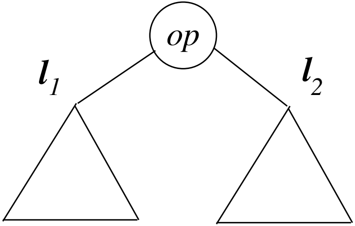
- Note:
  1. In instruction 3, the memory location is the right operand.
  2. In instruction 4, the destination register is the same as the left operand register.

Key Idea

- Determines an evaluation order of the subtrees which requires *minimum number of registers.*

Key Idea

- Determines an evaluation order of the subtrees which requires *minimum number of registers.*
- If the left and right subtrees require l_1, and l_2 ($l_1 < l_2$) registers respectively, what should be the order of evaluation?



Key Idea

- *Choice 1*

Key Idea

- *Choice 1*
  1. Evaluate left subtree first, leaving result in a register. This requires upto l_1 registers.

Key Idea

- *Choice 1*
    1. Evaluate left subtree first, leaving result in a register. This requires upto l_1 registers.
    2. Evaluate the right subtree. During this we might require upto $l_2 + 1$ registers (l_2 registers for evaluating the right subtree and one register to hold the value of the left subtree.)

Key Idea

- *Choice 1*
  1. Evaluate left subtree first, leaving result in a register. This requires upto l_1 registers.
  2. Evaluate the right subtree. During this we might require upto $l_2 + 1$ registers (l_2 registers for evaluating the right subtree and one register to hold the value of the left subtree.)
- The maximum register requirement in this case is $max(l_1, l_2 + 1) = l_2 + 1$.

Key Idea

- *Choice 2*

Key Idea

- *Choice 2*
  1. Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require upto l_2 registers.

Key Idea

- *Choice 2*
  1. Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require upto l_2 registers.
  2. Evaluate the left subtree. During this, we might require upto $l_1 + 1$ registers.

Key Idea

- *Choice 2*
  1. Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require upto l_2 registers.
  2. Evaluate the left subtree. During this, we might require upto $l_1 + 1$ registers.
- The maximum register requirement over the whole tree is

$$max(l_1 + 1,\ l_2) = l_2$$

Key Idea

- *Choice 2*
  1. Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require upto l_2 registers.
  2. Evaluate the left subtree. During this, we might require upto $l_1 + 1$ registers.
- The maximum register requirement over the whole tree is

$$max(l_1 + 1,\ l_2) = l_2$$

Key Idea

- *Choice 2*
  1. Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require upto l_2 registers.
  2. Evaluate the left subtree. During this, we might require upto $l_1 + 1$ registers.
- The maximum register requirement over the whole tree is

$$max(l_1 + 1,\ l_2) = l_2$$

Therefore the subtree requiring more registers should be evaluated first.

Labeling the Expression Tree

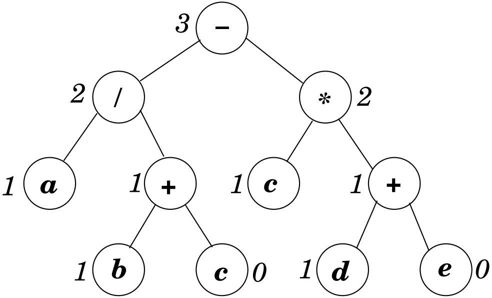
- Label each node by the number of registers required to evaluate it in a store free manner.

Labeling the Expression Tree

- Label each node by the number of registers required to evaluate it in a store free manner.

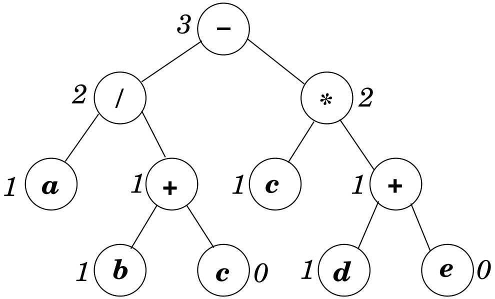
Labeling the Expression Tree

- Label each node by the number of registers required to evaluate it in a store free manner.



Labeling the Expression Tree

- Label each node by the number of registers required to evaluate it in a store free manner.



- Left and the right leaves are labeled 1 and 0 respectively, because the left leaf must necessarily be in a register, whereas the right leaf can reside in memory.

Labeling the Expression Tree

- Visit the tree in post-order. For every node visited do:

Labeling the Expression Tree

- Visit the tree in post-order. For every node visited do:
    1. Label each left leaf by 1 and each right leaf by 0.

Labeling the Expression Tree

- Visit the tree in post-order. For every node visited do:
  1. Label each left leaf by 1 and each right leaf by 0.
  2. If the labels of the children of a node n are l_1 and l_2 respectively, then

$$\begin{aligned} label(n) &= max(l_1, l_2), if\, l_1 \neq l_2 \\ &= l_1 + 1, otherwise \end{aligned}$$

Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function *gencode*(n), which produces code to evaluate the node labeled n.

Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function *gencode*(n), which produces code to evaluate the node labeled n.
2. Register allocation is done from a stack of register names *rstack*, initially containing $r_0, r_1, \ldots, r_k$ (with r_0 on top of the stack).

Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function *gencode*(n), which produces code to evaluate the node labeled n.
2. Register allocation is done from a stack of register names *rstack*, initially containing $r_0, r_1, \ldots, r_k$ (with r_0 on top of the stack).
3. *gencode*(n) evaluates n in the register on the top of the stack.

Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function *gencode*(n), which produces code to evaluate the node labeled n.
2. Register allocation is done from a stack of register names *rstack*, initially containing $r_0, r_1, \ldots, r_k$ (with r_0 on top of the stack).
3. *gencode*(n) evaluates n in the register on the top of the stack.
4. Temporary allocation is done from a stack of temporary names *tstack*, initially containing $t_0, t_1, \ldots, t_k$ (with t_0 on top of the stack).

Assumptions and Notational Conventions

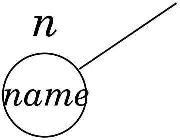
1. The code generation algorithm is represented as a function *gencode*(n), which produces code to evaluate the node labeled n.
2. Register allocation is done from a stack of register names *rstack*, initially containing $r_0, r_1, \ldots, r_k$ (with r_0 on top of the stack).
3. *gencode*(n) evaluates n in the register on the top of the stack.
4. Temporary allocation is done from a stack of temporary names *tstack*, initially containing $t_0, t_1, \ldots, t_k$ (with t_0 on top of the stack).
5. *swap*(*rstack*) swaps the top two registers on the stack.

The Algorithm

- *gencode*(n) described by case analysis on the type of the node n.

The Algorithm

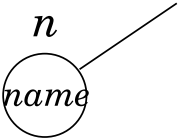
- *gencode*(n) described by case analysis on the type of the node n.
  1. *n is a left leaf:*



The Algorithm

- *gencode*(n) described by case analysis on the type of the node n.

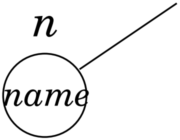
  1. *n is a left leaf:*



The Algorithm

- *gencode*(*n*) described by case analysis on the type of the node *n*.

    1. *n is a left leaf:*

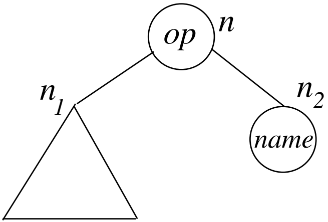


$$gen(top(rstack) \leftarrow name)$$

Comments: *n* is named by a variable say *name*. Code is generated to load *name* into a register.

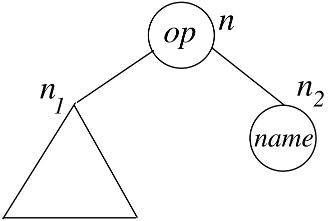
The Algorithm

2. *n's right child is a leaf:*



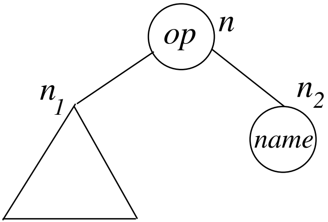
The Algorithm

2. *n's right child is a leaf:*



The Algorithm

2. *n's right child is a leaf:*



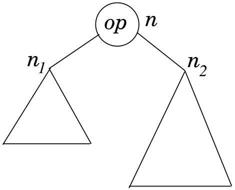
$gencode(n_1)$

$gen(top(rstack) \leftarrow top(rstack)\ op\ name)$

Comments: n_1 is first evaluated in the register on the top of the stack, followed by the operation *op* leaving the result in the same register.

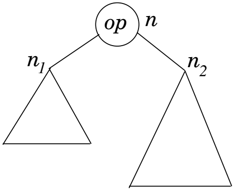
The Algorithm

3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



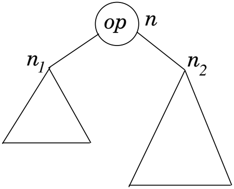
The Algorithm

3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



The Algorithm

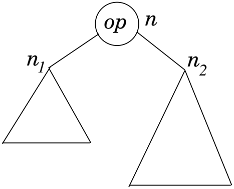
3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



swap(*rstack*); Right child goes into next to top register

The Algorithm

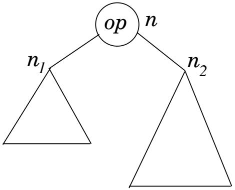
3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



swap(*rstack*);	Right child goes into next to top register
gencode(n_2);	Evaluate right child

The Algorithm

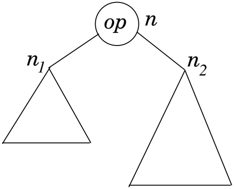
3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



swap(*rstack*);	Right child goes into next to top register
gencode(n_2);	Evaluate right child
$R := pop(rstack)$;	

The Algorithm

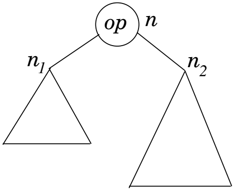
3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



$swap(rstack)$;	Right child goes into next to top register
$gencode(n_2)$;	Evaluate right child
$R := pop(rstack)$;	
$gencode(n_1)$;	Evaluate left child

The Algorithm

3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*

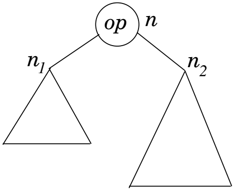


swap(*rstack*);	Right child goes into next to top register
gencode(n_2);	Evaluate right child
R := *pop*(*rstack*);	
gencode(n_1);	Evaluate left child

gen(*top*(*rstack*) ← *top*(*rstack*) *op* R); Issue *op*

The Algorithm

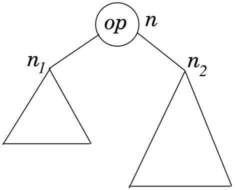
3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



$swap(rstack)$;	Right child goes into next to top register
$gencode(n_2)$;	Evaluate right child
$R := pop(rstack)$;	
$gencode(n_1)$;	Evaluate left child
$gen(top(rstack) \leftarrow top(rstack)\ op\ R)$;	Issue *op*
$push(rstack, R)$;	

The Algorithm

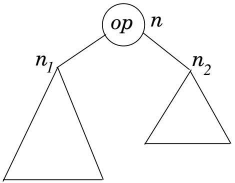
3. *The left child of n requires lesser number of registers. This requirement is strictly less than the available number of registers*



$swap(rstack)$;	Right child goes into next to top register
$gencode(n_2)$;	Evaluate right child
$R := pop(rstack)$;	
$gencode(n_1)$;	Evaluate left child
$gen(top(rstack) \leftarrow top(rstack)\ op\ R)$;	Issue *op*
$push(rstack, R)$;	
$swap(rstack)$	Restore register stack

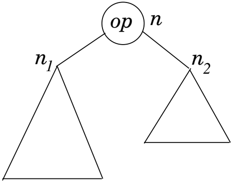
The Algorithm

4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



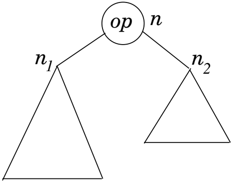
The Algorithm

4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



The Algorithm

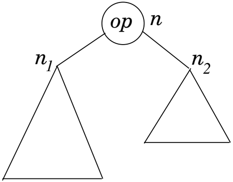
4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



gencode(n_1);

The Algorithm

4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*

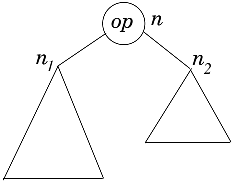


$gencode(n_1);$

$R := pop(rstack);$

The Algorithm

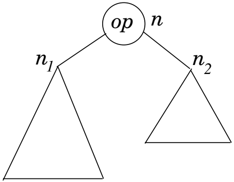
4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



$gencode(n_1);$
$R := pop(rstack);$
$gencode(n_2);$

The Algorithm

4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



$gencode(n_1);$

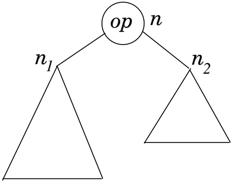
$R := pop(rstack);$

$gencode(n_2);$

$gen(R \leftarrow R\ op\ top(rstack));$

The Algorithm

4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



$gencode(n_1);$

$R := pop(rstack);$

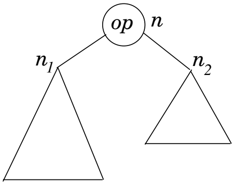
$gencode(n_2);$

$gen(R \leftarrow R\ op\ top(rstack));$

$push(rstack, R)$

The Algorithm

4. *The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*

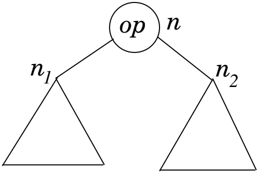


$gencode(n_1);$
$R := pop(rstack);$
$gencode(n_2);$
$gen(R \leftarrow R\ op\ top(rstack));$
$push(rstack, R)$

Comments: Same as case 3, except that the left sub-tree is evaluated first.

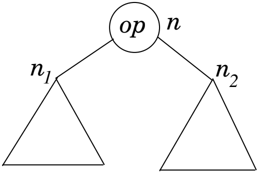
The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*



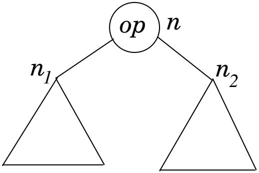
The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*



The Algorithm

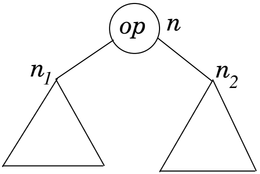
5. *Both the children of n require registers greater or equal to the available number of registers.*



$gencode(n_2);$

The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*

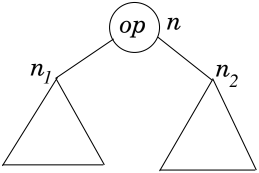


$gencode(n_2)$;

$T := pop(tstack)$;

The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*



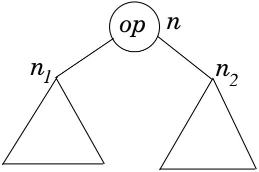
$gencode(n_2);$

$T := pop(tstack);$

$gen(T \leftarrow \; top(rstack));$

The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*



$gencode(n_2);$

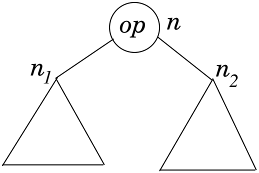
$T := pop(tstack);$

$gen(T \leftarrow\ top(rstack));$

$gencode(n_1);$

The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*



$gencode(n_2);$

$T := pop(tstack);$

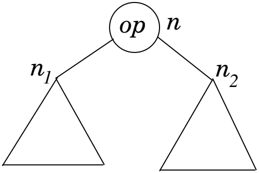
$gen(T \leftarrow \ top(rstack));$

$gencode(n_1);$

$push(tstack, T);$

The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*



$gencode(n_2);$

$T := pop(tstack);$

$gen(T \leftarrow\ top(rstack));$

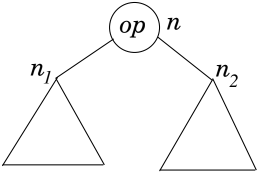
$gencode(n_1);$

$push(tstack, T);$

$gen(top(rstack) \leftarrow\ top(rstack)\ op\ T);$

The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*



$gencode(n_2);$

$T := pop(tstack);$

$gen(T \leftarrow\ top(rstack));$

$gencode(n_1);$

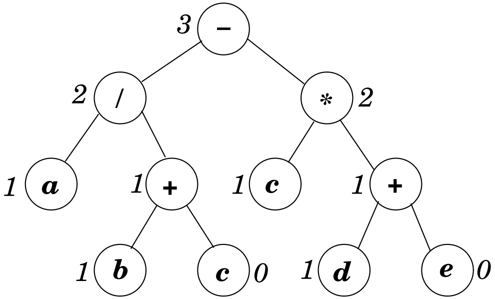
$push(tstack, T);$

$gen(top(rstack) \leftarrow\ top(rstack)\ op\ T);$

Comments: In this case the right sub-tree is first evaluated into a temporary. This is followed by the evaluations of the left sub-tree and *n* into the register on the top of the stack.

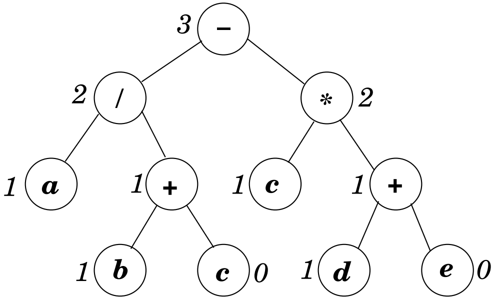
An Example

For the example:



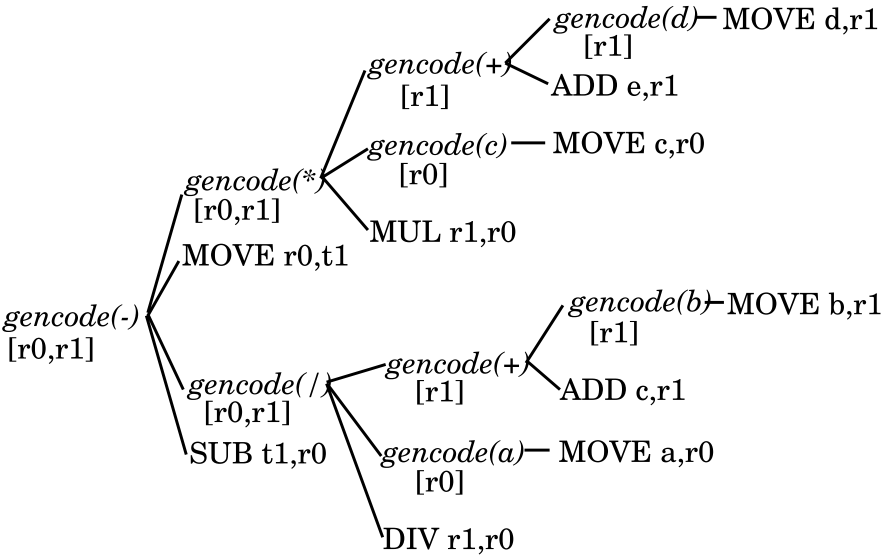
An Example

For the example:



assuming two available registers r_0 and r_1, the calls to gencode and the generated code are shown on the next slide.

An Example


gencode(-)
[r0,r1]
gencode(*)
[r0,r1]
MOVE r0,t1
gencode(/)
[r0,r1]
SUB t1,r0
gencode(+)
[r1]
gencode(c)
[r0]
MUL r1,r0
gencode(d)
[r1]
MOVE d,r1
ADD e,r1
MOVE c,r0
gencode(+)
[r1]
gencode(a)
[r0]
DIV r1,r0
gencode(b)
[r1]
MOVE b,r1
ADD c,r1
MOVE a,r0

SETHI-ULLMAN ALGORITHM: OPTIMALITY

- The algorithm is optimal because

SETHI-ULLMAN ALGORITHM: OPTIMALITY

- The algorithm is optimal because
  1. The number of load instructions generated is optimal.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

- The algorithm is optimal because
  1. The number of load instructions generated is optimal.
  2. Each binary operation specified in the expression tree is performed only once.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

- The algorithm is optimal because
  1. The number of load instructions generated is optimal.
  2. Each binary operation specified in the expression tree is performed only once.
  3. The number of stores is optimal.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

- The algorithm is optimal because
  1. The number of load instructions generated is optimal.
  2. Each binary operation specified in the expression tree is performed only once.
  3. The number of stores is optimal.
- We shall now elaborate on each of these.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

1. It is easy to verify that the number of loads required by any program computing an expression tree is at least equal to the number of left leaves. This algorithm generates no more loads than this.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

1. It is easy to verify that the number of loads required by any program computing an expression tree is at least equal to the number of left leaves. This algorithm generates no more loads than this.
2. Each node of the expression tree is visited exactly once. If this node specifies a binary operation, then the algorithm branches into steps 2,3,4 or 5, and at each of these places code is generated to perform this operation exactly once.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

3. The number of stores is optimal: this is harder to show.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

3. The number of stores is optimal: this is harder to show.
   - Define a *major node* as a node, each of whose children has a label at least equal to the number of available registers.

SETHI-ULLMAN ALGORITHM: OPTIMALITY

3. The number of stores is optimal: this is harder to show.
   - Define a *major node* as a node, each of whose children has a label at least equal to the number of available registers.
   - If we can show that the number of stores required by *any program* computing an expression tree is at least equal the number of major nodes, then our algorithm produces minimal number of stores (Why?)

SETHI-ULLMAN ALGORITHM

- To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.

SETHI-ULLMAN ALGORITHM

- To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- Assume that the tree has M major nodes.

SETHI-ULLMAN ALGORITHM

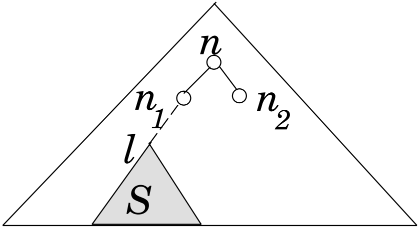
- To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- Assume that the tree has M major nodes.
- Now consider a tree formed by replacing the subtree S evaluated by the first store, with a leaf labeled by a name l.

SETHI-ULLMAN ALGORITHM

- To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- Assume that the tree has M major nodes.
- Now consider a tree formed by replacing the subtree S evaluated by the first store, with a leaf labeled by a name l.

SETHI-ULLMAN ALGORITHM

- To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- Assume that the tree has M major nodes.
- Now consider a tree formed by replacing the subtree S evaluated by the first store, with a leaf labeled by a name l.



- Let n be the major node in the original tree, just above S, and n_1 and n_2 be its immediate descendants (n_1 could be l itself).

SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of n_1 might have decreased but the label of n_2 remains unaffected ($\geq k$, the available number of registers).

SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of n_1 might have decreased but the label of n_2 remains unaffected ($\geq k$, the available number of registers).
2. The label of n is $\geq k$.

SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of n_1 might have decreased but the label of n_2 remains unaffected ($\geq k$, the available number of registers).
2. The label of n is $\geq k$.
3. The node n may no longer be a major node *but all other major nodes in the original tree continue to be major nodes in the modified tree*.

SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of n_1 might have decreased but the label of n_2 remains unaffected ($\geq k$, the available number of registers).
2. The label of n is $\geq k$.
3. The node n may no longer be a major node *but all other major nodes in the original tree continue to be major nodes in the modified tree*.
4. Therefore the number of major nodes in the modified tree is $M - 1$.

SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of n_1 might have decreased but the label of n_2 remains unaffected ($\geq k$, the available number of registers).
2. The label of n is $\geq k$.
3. The node n may no longer be a major node *but all other major nodes in the original tree continue to be major nodes in the modified tree*.
4. Therefore the number of major nodes in the modified tree is $M - 1$.
5. If we assume as induction hypothesis that the number of stores for the modified tree is at least $M - 1$, then the number of stores for the original tree is at least M.

SETHI-ULLMAN ALGORITHM: COMPLEXITY

Since the algorithm visits every node of the expression tree twice – once during labeling, and once during code generation, the complexity of the algorithm is $O(n)$.