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).

- 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 $a \times b$ has to be evaluated using $r_1 \leftarrow r_1 \times r_2$, 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.
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 \( a \times b \) has to be evaluated using \( r_1 \leftarrow r_1 \times r_2 \), 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.
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

- 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.
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 $a \times b$ has to be evaluated using $r_1 \leftarrow r_1 \times r_2$, 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.
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 $a \times b$ has to be evaluated using $r_1 \leftarrow r_1 \times r_2$, 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 $a \times b$ has to be evaluated using $r_1 \leftarrow r_1 \times r_2$, 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 $a \times b$ has to be evaluated using $r_1 \leftarrow r_1 \times r_2$, 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 $a \times b$ has to be evaluated using $r_1 \leftarrow r_1 \times r_2$, 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 \( \frac{a}{b + c} - c \cdot (d + e) \) represented as a tree:
Expression Trees

Here is the expression $a/(b + c) - c \times (d + e)$ represented as a tree:
Expression Trees

Here is the expression \( \frac{a}{(b + c)} - c \times (d + e) \) represented as a tree:
Expression Trees

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

```
/   *
\   /
  \ b
    +
      +
      +
      c
    d e
  a
```

Let $\Sigma$ be a countable set of variable names, and $\Theta$ be a finite set of binary operators. Then,

1. A single vertex labeled by a name from $\Sigma$ is an expression tree.
2. If $T_1$ and $T_2$ are expression trees and $\theta$ is an operator in $\Theta$, then $T_1 \theta T_2$ is an expression tree.

In this example $\Sigma = \{a, b, c, d, e, \ldots\}$, and $\Theta = \{+, -, \times, \div, \ldots\}$.
Expression Trees

Let $\Sigma$ be a countable set of variable names, and $\Theta$ be a finite set of binary operators. Then,

1. A single vertex labeled by a name from $\Sigma$ is an expression tree.
Expression Trees

Let \( \Sigma \) be a countable set of variable names, and \( \Theta \) be a finite set of binary operators. Then,

1. A single vertex labeled by a name from \( \Sigma \) is an expression tree.
2. If \( T_1 \) and \( T_2 \) are expression trees and \( \theta \) is a operator in \( \Theta \), then

\[
\begin{array}{c}
\theta \\
T_1 \\
T_2
\end{array}
\]

is an expression tree.
Expression Trees

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

\[
\text{then } \quad \begin{array}{c}
\theta \\
T_1 \\
T_2
\end{array}
\text{ is an expression tree.}
\]

- In this example

  \[\Sigma = \{a, b, c, d, e, \ldots\}, \text{ and } \Theta = \{+, -, *, /, \ldots\}\]
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. $r \leftarrow r \text{ op } m$ (the result of $r \text{ op } m$ is stored in $r$)
  4. $r_2 \leftarrow r_2 \text{ op } r_1$ (the result of $r_2 \text{ op } r_1$ 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.
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)
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. $m \leftarrow r$ \hspace{1cm} (store instruction)
  2. $r \leftarrow m$ \hspace{1cm} (load instruction)

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.
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 $r$)

- 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.
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. $r \leftarrow r \text{ op } m$ (the result of $r \text{ op } m$ is stored in $r$)
  4. $r_2 \leftarrow r_2 \text{ op } r_1$ (the result of $r_2 \text{ op } r_1$ 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.
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:

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.
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:

  1. In instruction 3, the memory location is the right operand.
Target Machine Model

We assume a machine with finite set of registers $r_0, r_1, \ldots, r_k$, 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:

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**

  1. Evaluate left subtree first, leaving result in a register. This requires up to \( l_1 \) registers.
  2. Evaluate the right subtree. During this we might require up to \( l_2 + 1 \) 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 1**
  1. Evaluate left subtree first, leaving result in a register. This requires up to $l_1$ 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 up to \( l_1 \) registers.
2. Evaluate the right subtree. During this we might require up to \( l_2 + 1 \) 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

1. Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require up to $l_2$ registers.

2. Evaluate the left subtree. During this, we might require up to $l_1 + 1$ 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.
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.

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.
Key Idea

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

2. Evaluate the left subtree. During this, we might require up to \( l_1 + 1 \) 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 up to $l_2$ registers.
  2. Evaluate the left subtree. During this, we might require up to $l_1 + 1$ 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.

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

- 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.

```
/  \
*  /
+  1
  1 0
  1 1
  1 1
  1 1
-  3
  2
```

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

- 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:
  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 label $(n) = \max(l_1, l_2)$, if $l_1 \neq l_2 = l_1 + 1$, otherwise
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

$$
label(n) = \begin{cases} 
max(l_1, l_2), & \text{if } l_1 \neq l_2 \\
 l_1 + 1, & \text{otherwise}
\end{cases}
$$
Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function $\text{gencode}(n)$, 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).

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.
Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function $\text{gencode}(n)$, 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. $\text{gencode}(n)$ 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 $\text{gencode}(n)$, 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. $\text{gencode}(n)$ 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).
1. The code generation algorithm is represented as a function $\text{gencode}(n)$, which produces code to evaluate the node labeled $n$.

2. Register allocation is done from a stack of register names $\text{rstack}$, initially containing $r_0, r_1, \ldots, r_k$ (with $r_0$ on top of the stack).

3. $\text{gencode}(n)$ evaluates $n$ in the register on the top of the stack.

4. Temporary allocation is done from a stack of temporary names $\text{tstack}$, initially containing $t_0, t_1, \ldots, t_k$ (with $t_0$ on top of the stack).

5. $\text{swap(rstack)}$ swaps the top two registers on the stack.
The Algorithm

- \textit{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:

```plaintext
    name
  gen(n) ← top(rstack)
```

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

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

\begin{figure}[h]
  \centering
  \begin{tikzpicture}
    \node [shape=circle,draw] (n) {\textit{n}};
    \node [shape=circle,draw, below of=n] (name) {\textit{name}};
    \draw [->] (n) -- (name);
  \end{tikzpicture}
\end{figure}

Comments: \textit{n} is named by a variable say \textit{name}. Code is generated to load \textit{name} into a register.
The Algorithm

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

1. $n$ is a left leaf:

   $$\text{gen}(\text{top}(rstack) \leftarrow \text{name})$$

Comments: $n$ is named by a variable say $\text{name}$. Code is generated to load $\text{name}$ into a register.
The Algorithm

2. *n’s right child is a leaf:*

![Diagram of a tree with nodes labeled as follows:](attachment:image.png)

\[
\text{gencode}(n) \rightarrow \text{gen}(\text{top}(	ext{rstack}) \leftarrow \text{top}(	ext{rstack}) \circ \text{name})
\]

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

2. *n’s right child is a leaf:*

\[ gencode(n_1) \]

\[ \text{gen}(\text{top}(\text{rstack}) \leftarrow \text{top}(\text{rstack}) \text{op} \text{name}) \]

Comments: \( n_1 \) is first evaluated in the register on the top of the stack, followed by the operation \( \text{op} \) leaving the result in the same register.
The Algorithm

2. *n’s right child is a leaf:*

```
gencode(n_1)
gen(top(rstack) ← 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*
3. The left child of \( n \) requires lesser number of registers. This requirement is strictly less than the available number of registers.
3. The left child of \( n \) requires lesser number of registers. This requirement is strictly less than the available number of registers.

\[
\text{swap(rstack)};
\]

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.

\[
\begin{align*}
\text{Evaluate left child:} & \quad \text{gen}(\text{top}(rstack)) \leftarrow \text{top}(rstack) \text{\( op \)} R \\
\text{Issue:} & \quad \text{push}(rstack, R) \\
\text{swap}(rstack); & \quad \text{Right child goes into next to top register} \\
\text{gen}(n_2); & \quad \text{Evaluate right child}
\end{align*}
\]
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.

\[
\begin{align*}
\text{swap}(rstack); & \quad \text{Right child goes into next to top register} \\
gencode(n_2); & \quad \text{Evaluate right child} \\
R := \text{pop}(rstack); & \\
gencode(n_1); & \quad \text{Evaluate left child}
\end{align*}
\]
The Algorithm

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

\[
\begin{align*}
\text{swap}(rstack); & \quad \text{Right child goes into next to top register} \\
gencode(n_2); & \quad \text{Evaluate right child} \\
R := \text{pop}(rstack); & \\
gencode(n_1); & \quad \text{Evaluate left child} \\
gen(top(rstack) \leftarrow top(rstack) \ op \ R); & \quad \text{Issue op}
\end{align*}
\]
The Algorithm

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

![Tree Diagram]

\[
\text{swap}(\text{rstack}); \quad \text{Right child goes into next to top register}
\]

\[
\text{gencode}(n_2); \quad \text{Evaluate right child}
\]

\[
R \leftarrow \text{pop}(\text{rstack});
\]

\[
\text{gencode}(n_1); \quad \text{Evaluate left child}
\]

\[
\text{gen}(\text{top}(\text{rstack}) \leftarrow \text{top}(\text{rstack}) \ op \ R); \quad \text{Issue op}
\]

\[
\text{push}(\text{rstack}, R);
\]
The Algorithm

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

\[
\begin{align*}
\text{swap}(rstack); & \quad \text{Right child goes into next to top register} \\
gencode(n_2); & \quad \text{Evaluate right child} \\
R := \text{pop}(rstack); & \\
gencode(n_1); & \quad \text{Evaluate left child} \\
gen(\text{top}(rstack) \leftarrow \text{top}(rstack) \ op \ R); & \quad \text{Issue } op \\
push(rstack, R); & \\
\text{swap}(rstack) & \quad \text{Restore register stack}
\end{align*}
\]
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.

\[
\begin{align*}
\text{op} & \quad n \\
& \quad n_1 \\
& \quad n_2
\end{align*}
\]
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.

\[
gencode(n_1);
\]

\[
gen(R \leftarrow R \ op \ top(rstack));
\]

\[
pop(rstack);\]

\[
gencode(n_2);
\]

\[
push(rstack, R)
\]
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(n1);
R := pop(rstack);
```
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

\[
gencode(n_1); \\
R := \text{pop(rstack)}; \\
gencode(n_2);
\]
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 ← R op top(rstack));
push(rstack, R)
```
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 ← 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.

$$\text{gencode}(n_2); T := \text{pop}(tstack); \text{gen}(T \leftarrow \text{top}(rstack)); \text{gencode}(n_1); \text{push}(tstack, T); \text{gen}(\text{top}(rstack) \leftarrow \text{top}(rstack) \text{op } T);$$

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.
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.*

\[
\begin{align*}
\text{gencode}(n_2); & \\
\end{align*}
\]
The Algorithm

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

```
\[\text{gen}(T \leftarrow \text{top}(rstack));
\]
\[\text{gencode}(n_1);
\]
\[T := \text{pop}(tstack);
\]
\[\text{gencode}(n_2);
\]
```

Diagram:
```
  op
   /\  n
  /   /
n_1  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);
gen(T ← top(rstack));
```
The Algorithm

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

![Diagram of a tree with root $op$, left child $n_1$, and right child $n_2$]

```
gencode(n_2);
T := pop(tstack);
gen(T ← 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 ← top(rstack));
gencode(n_1);
push(tstack, T);
```
5. Both the children of \( n \) require registers greater or equal to the available number of registers.

\[
gencode(\ n_2\ );
T := \text{pop}(tstack);
gen(T \leftarrow \text{top}(rstack));
gencode(\ n_1\ );
push(tstack, T);
gen(\text{top}(rstack) \leftarrow \text{top}(rstack) \text{ op } T);\]
The Algorithm

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

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(/)
SUB t1,r0
gencode(*)
MOVE r0,t1
[r0,r1]
[r0,r1]
gencode(-)
[r0,r1]
gencode(+)
MUL r1,r0
gencode(a)
ADD e,r1
[r0]
gencode(c)
MOVE c,r0
MUL r1,r0
gencode(b)
ADD c,r1
[r1]
gencode(d)
MOVE d,r1
[r1]
gencode(+)
ADD e,r1
[r1]
DIV r1,r0
MOV b,r1
[r1]
gencode(a)
MOVE a,r0
[r0]
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.
The algorithm is optimal because

1. The number of load instructions generated is optimal.
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.
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.
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.
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.
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.
3. The number of stores is optimal: this is harder to show.
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.
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.
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 $l$. 
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).
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 \( M - 1 \).
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 \( M \).
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 $O(n)$. 