Code Generation: Aho Johnson Algorithm

Amey Karkare
karkare@cse.iitk.ac.in

April 5, 2019

Aho-Johnson Algorithm

Characteristics of the Algorithm

- Considers expression trees.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height
- Does not use algebraic properties of operators.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height
- Does not use algebraic properties of operators.
- Generates optimal code, where, once again, the cost measure is the number of instructions in the code.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height
- Does not use algebraic properties of operators.
- Generates optimal code, where, once again, the cost measure is the number of instructions in the code.
- Complexity is linear in the size of the expression tree.

Expression Trees Defined

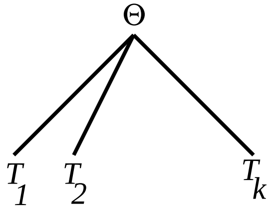
- Let Σ be a countable set of operands, and Θ be a finite set of operators. Then,

Expression Trees Defined

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

Expression Trees Defined

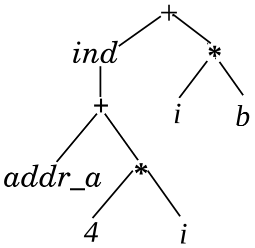
- Let Σ be a countable set of operands, and Θ be a finite set of operators. Then,
    1. A single vertex labeled by a name from Σ is an expression tree.
    2. If T_1, T_2, ..., T_k are expression trees whose leaves all have distinct labels and θ is a k-ary operator in Θ, then



is an expression tree.

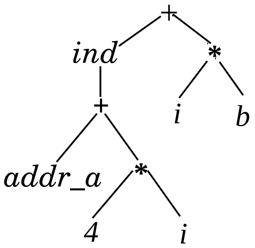
Example

- An example of an expression tree is



Example

- An example of an expression tree is



- **Notation:** If T is an expression tree, and S is a subtree of T, then T/S is the the tree obtained by replacing S in T by a single leaf labeled by a distinct name from Σ.

The Machine Model

1. The machine has *n* general purpose registers (no special registers).

The Machine Model

1. The machine has n general purpose registers (no special registers).
2. Countable sequence of memory locations.

The Machine Model

1. The machine has n general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:

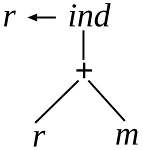
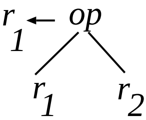
The Machine Model

1. The machine has n general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:
   a. $r \leftarrow E$, r is a register and E is an expression tree whose operators are from Θ and operands are registers, memory locations or constants. Further, *r should be one of the register names occurring (if any) in E*.

The Machine Model

1. The machine has n general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:
    a. $r \leftarrow E$, r is a register and E is an expression tree whose operators are from Θ and operands are registers, memory locations or constants. Further, *r should be one of the register names occurring (if any) in E.*
    b. $m \leftarrow r$, a store instruction.

Example Of A Machine

$r \leftarrow c$	**{MOV #c, r}**
$r \leftarrow m$	**{MOV m, r}**
$m \leftarrow r$	**{MOV r, m}**
	**{MOV m(r), r}**
	**{op r_2, r_1}**

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \dots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \dots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \dots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$
$$r_2 \leftarrow ind(r_2)$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$
\begin{aligned}
r_1 &\leftarrow 4 \\
r_1 &\leftarrow r_1 * i \\
r_2 &\leftarrow addr\_a \\
r_2 &\leftarrow r_2 + r_1 \\
r_2 &\leftarrow ind(r_2) \\
r_3 &\leftarrow i
\end{aligned}
$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$
$$r_2 \leftarrow ind(r_2)$$
$$r_3 \leftarrow i$$
$$r_3 \leftarrow r_3 * b$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$
$$r_2 \leftarrow ind(r_2)$$
$$r_3 \leftarrow i$$
$$r_3 \leftarrow r_3 * b$$
$$r_2 \leftarrow r_2 + r_3$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$
\begin{aligned}
r_1 &\leftarrow 4 \\
r_1 &\leftarrow r_1 * i \\
r_2 &\leftarrow addr\_a \\
r_2 &\leftarrow r_2 + r_1 \\
r_2 &\leftarrow ind(r_2) \\
r_3 &\leftarrow i \\
r_3 &\leftarrow r_3 * b \\
r_2 &\leftarrow r_2 + r_3
\end{aligned}
$$

VALUE OF A PROGRAM

- We need to define the value $v(P)$ computed by a program P.

VALUE OF A PROGRAM

- We need to define the value $v(P)$ computed by a program P.
    1. We want to specify what it means to say that a *program P computes an expression tree T*. This is when the value of the program $v(P)$ is the same as T.

VALUE OF A PROGRAM

- We need to define the value $v(P)$ computed by a program P.
  1. We want to specify what it means to say that a *program P computes an expression tree T*. This is when the value of the program $v(P)$ is the same as T.
  2. We also want to talk of *equivalence* of two programs P_1 and P_2. This is true when $v(P_1) = v(P_2)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
    a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
  b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
  b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.
  c. If I_t is $m \leftarrow r$, then $v_t(m)$ is $v_{t-1}(r)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
    a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
    b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.
    c. If I_t is $m \leftarrow r$, then $v_t(m)$ is $v_{t-1}(r)$.
    d. Otherwise $v_t(z) = v_{t-1}(z)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
  b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.
  c. If I_t is $m \leftarrow r$, then $v_t(m)$ is $v_{t-1}(r)$.
  d. Otherwise $v_t(z) = v_{t-1}(z)$.
- If I_q is $z \leftarrow E$, then the value of P is $v_q(z)$.

EXAMPLE

- For the program:

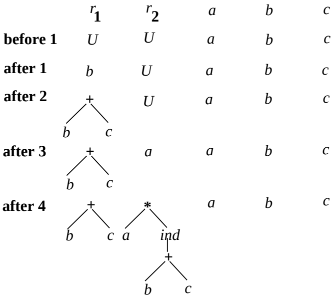
$$r_1 \leftarrow b$$
$$r_1 \leftarrow r_1 + c$$
$$r_2 \leftarrow a$$
$$r_2 \leftarrow r_2 * ind(r_1)$$

EXAMPLE

- For the program:

$$r_1 \leftarrow b$$
$$r_1 \leftarrow r_1 + c$$
$$r_2 \leftarrow a$$
$$r_2 \leftarrow r_2 * ind(r_1)$$

- the values of r_1, r_2, a, b and c at different time instants are:

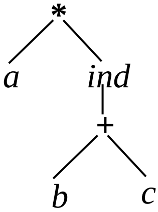


EXAMPLE

- For the program:

$$r_1 \leftarrow b$$
$$r_1 \leftarrow r_1 + c$$
$$r_2 \leftarrow a$$
$$r_2 \leftarrow r_2 * ind(r_1)$$

- The values of of the program is



USELESS INSTRUCTIONS

- An instruction I_t in a program P is said to be *useless*, if the program P_1 formed by removing I_t from P is equivalent to P.

USELESS INSTRUCTIONS

- An instruction I_t in a program P is said to be *useless*, if the program P_1 formed by removing I_t from P is equivalent to P.
- NOTE: We shall assume that our programs do not have any useless instructions.

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \dots I_q$ is the sequence of instructions $I_{t+1}, \dots, I_s$, where s is the largest index such that

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \ldots I_q$ is the sequence of instructions $I_{t+1}, \ldots, I_s$, where s is the largest index such that

  a. The register or memory location defined by I_t is used by I_s, and

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \ldots I_q$ is the sequence of instructions $I_{t+1}, \ldots, I_s$, where s is the largest index such that
  a. The register or memory location defined by I_t is used by I_s, and
  b. This register/memory location is not redefined by the instructions between I_t and I_s.

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \ldots I_q$ is the sequence of instructions $I_{t+1}, \ldots, I_s$, where s is the largest index such that
  a. The register or memory location defined by I_t is used by I_s, and
  b. This register/memory location is not redefined by the instructions between I_t and I_s.
- The relation between I_s and I_t is expressed by saying that I_s is the *last use* of I_t, and is denoted by $s = U_p(t)$.

REARRANGABILITY OF PROGRAMS

- We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.

REARRANGABILITY OF PROGRAMS

- We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.
- Why is this result important? This is because our algorithm considers programs which are in strong normal form only. The above result assures us that by doing so, we shall not miss out an optimal solution.

REARRANGABILITY OF PROGRAMS

- We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.
- Why is this result important? This is because our algorithm considers programs which are in strong normal form only. The above result assures us that by doing so, we shall not miss out an optimal solution.
- To show the above result, we shall have to consider the kinds of rearrangements which retain program equivalence.

Rearrangement Theorem

- Let $P = I_1, I_2, \ldots, I_q$ be a program which computes an expression tree.

Rearrangement Theorem

- Let $P = I_1, I_2, \ldots, I_q$ be a program which computes an expression tree.
- Let π be a permutation on $\{1 \ldots q\}$ with $\pi(q) = q$.

Rearrangement Theorem

- Let $P = I_1, I_2, \ldots, I_q$ be a program which computes an expression tree.
- Let π be a permutation on $\{1 \ldots q\}$ with $\pi(q) = q$.
- π induces a rearranged program $Q = J_1, J_2, \ldots, J_q$ with I_i in P becoming $J_{\pi(i)}$ in Q.

Rearrangement Theorem

- Let $P = I_1, I_2, \ldots, I_q$ be a program which computes an expression tree.
- Let π be a permutation on $\{1 \ldots q\}$ with $\pi(q) = q$.
- π induces a rearranged program $Q = J_1, J_2, \ldots, J_q$ with I_i in P becoming $J_{\pi(i)}$ in Q.
- Then Q is equivalent to P if $\pi(U_P(t)) = U_Q(\pi(t))$.

Rearrangement Theorem: Notes

- The rearrangement theorem merely states that a rearrangement retains program equivalence, if any variable defined by an instruction in the original program is last used by the same instructions in both the original and rearranged program.

Rearrangement Theorem: Notes

- The rearrangement theorem merely states that a rearrangement retains program equivalence, if any variable defined by an instruction in the original program is last used by the same instructions in both the original and rearranged program.
- To see why the statement of the theorem is true, reason as follows.

Rearrangement Theorem: Notes

a. P is equivalent to Q, if the operands used by the last instruction I_q(also J_q) have the same value in P and Q.

Rearrangement Theorem: Notes

a. P is equivalent to Q, if the operands used by the last instruction I_q(also J_q) have the same value in P and Q.

b. Consider any operand in I_q, say z. By the rearrangement theorem, This must have been defined by the same instruction (though in different positions say I_t and $J_{\pi(t)}$) in P and Q. So z in I_q and J_q have the same value, if the operands used by I_t and $J_{\pi(t)}$ have the same value in P and Q.

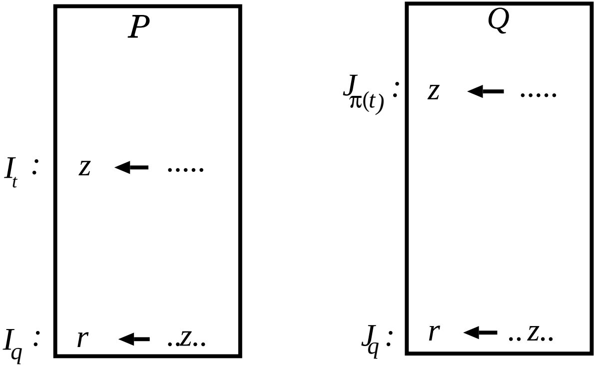
Rearrangement Theorem: Notes

a. P is equivalent to Q, if the operands used by the last instruction I_q(also J_q) have the same value in P and Q.

b. Consider any operand in I_q, say z. By the rearrangement theorem, This must have been defined by the same instruction (though in different positions say I_t and $J_{\pi(t)}$) in P and Q. So z in I_q and J_q have the same value, if the operands used by I_t and $J_{\pi(t)}$ have the same value in P and Q.

c. Repeat this argument, till you come across an instruction with all constants on the right hand side.

Rearrangement Theorem: Notes



WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \leq j \leq t$, with $U_P(j) > t$, and I_j not a store instruction.

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \leq j \leq t$, with $U_P(j) > t$, and I_j not a store instruction.

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \leq j \leq t$, with $U_P(j) > t$, and I_j not a store instruction.

$$\begin{array}{ll} & r_1 \leftarrow \\ & r_2 \leftarrow \\ I_t : & \qquad\qquad \text{Width} = 2 \\ & \leftarrow r_1 \\ & \leftarrow r_2 \end{array}$$

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \leq j \leq t$, with $U_P(j) > t$, and I_j not a store instruction.

$$\begin{array}{llll} & r_1 \leftarrow & & \\ & r_2 \leftarrow & & \\ I_t : & & & \text{Width} = 2 \\ & \leftarrow r_1 & & \\ & \leftarrow r_2 & & \end{array}$$

- The *width of a program* P is the maximum width over all instructions in P.

WIDTH

- A program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers.

WIDTH

- A program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers.
- EXAMPLE:

$$\begin{aligned} r_1 &\leftarrow a \\ r_2 &\leftarrow b \\ r_1 &\leftarrow r_1 + r_2 \\ r_3 &\leftarrow c \\ r_3 &\leftarrow r_3 + d \\ r_1 &\leftarrow r_1 * r_3 \end{aligned}$$

WIDTH

- A program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers.
- EXAMPLE:

$$\begin{aligned} r_1 &\leftarrow a \\ r_2 &\leftarrow b \\ r_1 &\leftarrow r_1 + r_2 \\ r_3 &\leftarrow c \\ r_3 &\leftarrow r_3 + d \\ r_1 &\leftarrow r_1 * r_3 \end{aligned} \qquad \begin{aligned} r_1 &\leftarrow a \\ r_2 &\leftarrow b \\ r_1 &\leftarrow r_1 + r_2 \\ r_2 &\leftarrow c \\ r_2 &\leftarrow r_2 + d \\ r_1 &\leftarrow r_1 * r_2 \end{aligned}$$

WIDTH

- A program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers.
- EXAMPLE:

$$\begin{aligned} r_1 &\leftarrow a \\ r_2 &\leftarrow b \\ r_1 &\leftarrow r_1 + r_2 \\ r_3 &\leftarrow c \\ r_3 &\leftarrow r_3 + d \\ r_1 &\leftarrow r_1 * r_3 \end{aligned} \qquad \begin{aligned} r_1 &\leftarrow a \\ r_2 &\leftarrow b \\ r_1 &\leftarrow r_1 + r_2 \\ r_2 &\leftarrow c \\ r_2 &\leftarrow r_2 + d \\ r_1 &\leftarrow r_1 * r_2 \end{aligned}$$

- In the example above, the first program has width 2 but uses 3 registers. By suitable renaming, the number of registers in the second program has been brought down to 2.

LEMMA

Let P be a program of width w, and let R be a set of w distinct registers. Then, by renaming the registers used by P, we may construct an equivalent program P', with the same length as P, which uses only registers in R.

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
    a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
   a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).
   b. There is no question of choice for the register r in the instruction $r \leftarrow E$, where E has some register operands. r has to be one of the registers occurring in E.

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
   a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).
   b. There is no question of choice for the register r in the instruction $r \leftarrow E$, where E has some register operands. r has to be one of the registers occurring in E.
   c. The only instructions involving a choice of registers are instructions of the form $r \leftarrow E$, where E has no register operands.

PROOF OUTLINE

3. Since the width of P is w, the width of the instruction just before $r \leftarrow E$ is at most $w - 1$. (Why?)

PROOF OUTLINE

3. Since the width of P is w, the width of the instruction just before $r \leftarrow E$ is at most $w - 1$. (Why?)
4. Therefore a register can always be found for r in the rearranged program P'.

CONTIGUITY AND STRONG CONTIGUITY

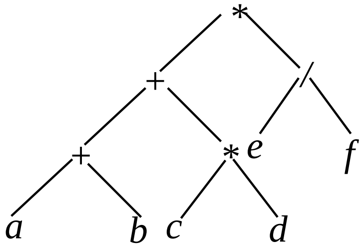
- Can one decrease the width of a program?

CONTIGUITY AND STRONG CONTIGUITY

- Can one decrease the width of a program?
- For *storeless programs*, there is an arrangement which has minimum width.

CONTIGUITY AND STRONG CONTIGUITY

- Can one decrease the width of a program?
- For *storeless programs*, there is an arrangement which has minimum width.
- EXAMPLE: All the three programs P_1, P_2, and P_3 compute the expression tree shown below:

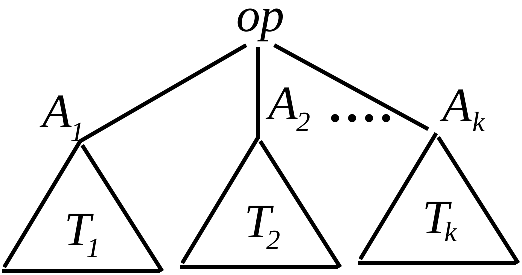


$\underline{P_1}$	$\underline{P_2}$	$\underline{P_3}$
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_3 \leftarrow c$	$r_3 \leftarrow c$	$r_1 \leftarrow r_1 + r_2$
$r_4 \leftarrow d$	$r_4 \leftarrow d$	$r_2 \leftarrow c$
$r_5 \leftarrow e$	$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow d$
$r_6 \leftarrow f$	$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow r_2 * r_3$
$r_5 \leftarrow r_5 / r_6$	$r_1 \leftarrow r_1 + r_3$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow e$	$r_2 \leftarrow e$
$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow f$	$r_3 \leftarrow f$
$r_1 \leftarrow r_1 + r_3$	$r_2 \leftarrow r_2 / r_3$	$r_2 \leftarrow r_2 / r_3$
$r_1 \leftarrow r_1 * r_5$	$r_1 \leftarrow r_1 * r_2$	$r_1 \leftarrow r_1 * r_2$

P_1	P_2	P_3
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_3 \leftarrow c$	$r_3 \leftarrow c$	$r_1 \leftarrow r_1 + r_2$
$r_4 \leftarrow d$	$r_4 \leftarrow d$	$r_2 \leftarrow c$
$r_5 \leftarrow e$	$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow d$
$r_6 \leftarrow f$	$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow r_2 * r_3$
$r_5 \leftarrow r_5 / r_6$	$r_1 \leftarrow r_1 + r_3$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow e$	$r_2 \leftarrow e$
$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow f$	$r_3 \leftarrow f$
$r_1 \leftarrow r_1 + r_3$	$r_2 \leftarrow r_2 / r_3$	$r_2 \leftarrow r_2 / r_3$
$r_1 \leftarrow r_1 * r_5$	$r_1 \leftarrow r_1 * r_2$	$r_1 \leftarrow r_1 * r_2$

The program P_2 has a width less than P_1, whereas P_3 has the least width of all three programs. P_2 is a *contiguous* program whereas P_3 is a *strongly contiguous* program.

CONTIGUITY AND STRONG CONTIGUITY



CONTIGUITY AND STRONG CONTIGUITY

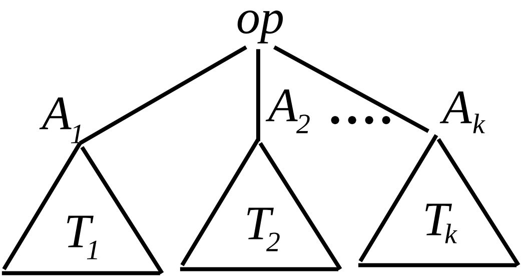
THEOREM: Let $P = I_1, I_2, \ldots, I_q$ be a program of width w with no stores. I_q uses k registers whose values at time $q-1$ are $A_1, \ldots, A_k$. Then there exists an equivalent program $Q = J_1, J_2, \ldots, J_q$, and a permutation π on $\{1, \ldots, k\}$ such that

i. Q has width at most w.

ii. Q can be written as $P_1 \ldots P_k J_q$ where $v(P_i) = A_{\pi(i)}$ for $1 \leq i \leq k$, and the width of P_i, by itself, is at most $w - i + 1$.

CONTIGUITY AND STRONG CONTIGUITY

Consider an evaluation of the expression tree:.



This tree can be evaluated in the order mentioned below:

CONTIGUOUS AND STRONG CONTIGUOUS EVALUATION

1. Q computes the entire subtree T_1 first using P_1. In the process all the w registers could be used.
2. After computing T_1 all registers except one are freed. Therefore T_2 is free to use $w - 1$ registers and its width is at most $w - 1$. T_2 is computed by P_2.
3. T_3 is similarly computed by P_3, whose width is $w - 2$.

Of course $A_1, \ldots, A_3$ need not necessarily be computed in this order. This is what brings the permutation π in the statement of the theorem.

CONTIGUOUS AND STRONG CONTIGUOUS EVALUATION

A program in the form $P_1 \ldots P_k J_q$ is said to be in *contiguous form*. If each of the P_is is, in turn, contiguous, then the program is said to be in *strong contiguous form*.
THEOREM: Every program *without stores* can be transformed into strongly contiguous form.
PROOF OUTLINE: Apply the technique in the previous theorem recursively to each of the P_is.

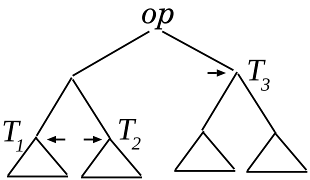
AHO-JOHNSON ALGORITHM

STRONG NORMAL FORM PROGRAMS

A program requires stores if there are not enough registers to hold intermediate values or if an instruction requires some of its operands to be in memory locations. Such programs can also be cast in a certain form called *strong normal form*.

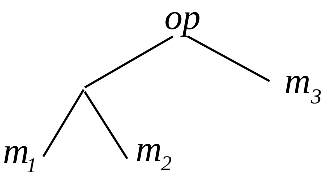
AHO-JOHNSON ALGORITHM

Consider the following evaluation of tree shown, in which the marked nodes require stores.



1. Compute T_1 using program P_1. Store the value in memory location m_1.
2. Compute T_2 using program P_2. Store the value in memory location m_2.
3. Compute T_3 using program P_3. Store the value in memory location m_3.
4. Compute the tree shown below using a storeless program P_4.

AHO-JOHNSON ALGORITHM



A program in such a form is called a *normal form program*.

AHO-JOHNSON ALGORITHM

Let $P = I_1 \dots I_q$ be a machine program. We say P is in *normal form*, if it can be written as $P = P_1 J_1 P_2 J_2 \dots P_{s-1} J_{s-1} P_s$, such that

1. Each J_i is a store instruction and no P_i contains a store instruction.
2. No registers are active immediately after a store instruction.

Further, P is in *strong normal form*, if each P_i is strongly contiguous.

AHO-JOHNSON ALGORITHM

LEMMA: Let P be an optimal program which computes an expression tree. Then there exists a permutation of P, which computes the same value and is in normal form.

AHO-JOHNSON ALGORITHM

LEMMA: Let P be an optimal program which computes an expression tree. Then there exists a permutation of P, which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let I_f be the first store instruction of P.

AHO-JOHNSON ALGORITHM

LEMMA: Let P be an optimal program which computes an expression tree. Then there exists a permutation of P, which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let I_f be the first store instruction of P.
2. Identify the instructions between I_1 and I_{f-1} which do not contribute towards the computation of the value of I_f.

AHO-JOHNSON ALGORITHM

LEMMA: Let P be an optimal program which computes an expression tree. Then there exists a permutation of P, which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let I_f be the first store instruction of P.
2. Identify the instructions between I_1 and I_{f-1} which do not contribute towards the computation of the value of I_f.
3. Shift these instructions, in order, after I_f.

AHO-JOHNSON ALGORITHM

LEMMA: Let P be an optimal program which computes an expression tree. Then there exists a permutation of P, which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let I_f be the first store instruction of P.
2. Identify the instructions between I_1 and I_{f-1} which do not contribute towards the computation of the value of I_f.
3. Shift these instructions, in order, after I_f.
4. We now have a program $P_1 J_1 Q$, where P_1 is storeless, J_1 is the first store instruction (previously denoted by I_f), and no registers are active after J_1.

AHO-JOHNSON ALGORITHM

LEMMA: Let P be an optimal program which computes an expression tree. Then there exists a permutation of P, which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let I_f be the first store instruction of P.
2. Identify the instructions between I_1 and I_{f-1} which do not contribute towards the computation of the value of I_f.
3. Shift these instructions, in order, after I_f.
4. We now have a program $P_1 J_1 Q$, where P_1 is storeless, J_1 is the first store instruction (previously denoted by I_f), and no registers are active after J_1.
5. Repeat this for the program Q.

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

1. P and Q have the same length.

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

1. P and Q have the same length.
2. Q has width at most w, and

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

1. P and Q have the same length.
2. Q has width at most w, and
3. Q is in strong normal form.

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

1. P and Q have the same length.
2. Q has width at most w, and
3. Q is in strong normal form.

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

1. P and Q have the same length.
2. Q has width at most w, and
3. Q is in strong normal form.

PROOF OUTLINE:

1. Given a program, first apply the previous lemma to get a program in normal form.

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

1. P and Q have the same length.
2. Q has width at most w, and
3. Q is in strong normal form.

PROOF OUTLINE:

1. Given a program, first apply the previous lemma to get a program in normal form.
2. Convert each P_i to strongly contiguous form.

AHO-JOHNSON ALGORITHM

THEOREM: Let P be an optimal program of width w. We can transform P into an equivalent program Q such that:

1. P and Q have the same length.
2. Q has width at most w, and
3. Q is in strong normal form.

PROOF OUTLINE:

1. Given a program, first apply the previous lemma to get a program in normal form.
2. Convert each P_i to strongly contiguous form.
3. None of the above transformations increase the width or length of the program.

AHO-JOHNSON ALGORITHM

OPTIMALITY CONDITION

Not all programs in strong normal form are optimal. We need to specify under what conditions is a program in strong normal form optimal. This will allow us later to prove the optimality of our code generation algorithm.

AHO-JOHNSON ALGORITHM

OPTIMALITY CONDITION

Not all programs in strong normal form are optimal. We need to specify under what conditions is a program in strong normal form optimal. This will allow us later to prove the optimality of our code generation algorithm.

1. If an expression tree can be evaluated without stores, then the optimal program will do so. Moreover it will use minimal number of instructions for this purpose.

AHO-JOHNSON ALGORITHM

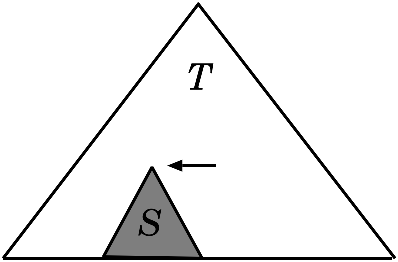
OPTIMALITY CONDITION

Not all programs in strong normal form are optimal. We need to specify under what conditions is a program in strong normal form optimal. This will allow us later to prove the optimality of our code generation algorithm.

1. If an expression tree can be evaluated without stores, then the optimal program will do so. Moreover it will use minimal number of instructions for this purpose.
2. Now assume that a program necessarily requires stores at certain points of the tree, as shown next. For simplicity, assume that this is the only store required to evaluate the tree.

AHO-JOHNSON ALGORITHM

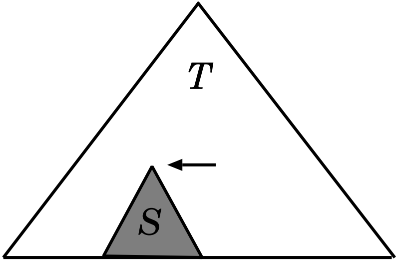
OPTIMALITY CONDITION



3. then the optimal program should

AHO-JOHNSON ALGORITHM

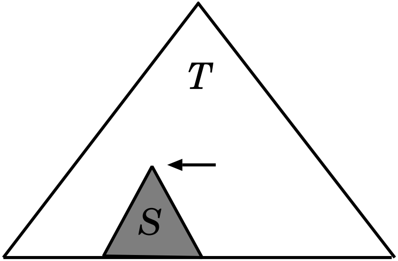
OPTIMALITY CONDITION



3. then the optimal program should
   a. Evaluate S (optimally, by condition 1).

AHO-JOHNSON ALGORITHM

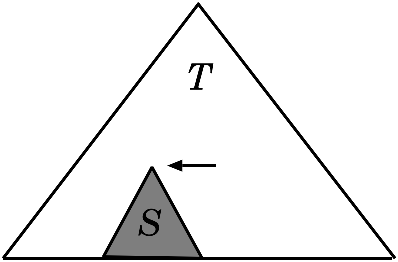
OPTIMALITY CONDITION



3. then the optimal program should
   a. Evaluate S (optimally, by condition 1).
   b. Store the value in a memory location.

AHO-JOHNSON ALGORITHM

OPTIMALITY CONDITION



3. then the optimal program should
   a. Evaluate S (optimally, by condition 1).
   b. Store the value in a memory location.
   c. Evaluate the rest of the (storeless) tree T/S (once again optimally, due to condition 1).

AHO-JOHNSON ALGORITHM

THE ALGORITHM

The algorithm makes three passes over the expression tree.

Pass 1 Computes an array of costs for each node. This helps to select an instruction to evaluate the node, and the evaluation order to evaluate the subtrees of the node.

AHO-JOHNSON ALGORITHM

THE ALGORITHM

The algorithm makes three passes over the expression tree.

Pass 1 Computes an array of costs for each node. This helps to select an instruction to evaluate the node, and the evaluation order to evaluate the subtrees of the node.

Pass 2 Identifies the subtrees which must be evaluated in memory locations.

AHO-JOHNSON ALGORITHM

THE ALGORITHM

The algorithm makes three passes over the expression tree.

Pass 1
: Computes an array of costs for each node. This helps to select an instruction to evaluate the node, and the evaluation order to evaluate the subtrees of the node.

Pass 2
: Identifies the subtrees which must be evaluated in memory locations.

Pass 3
: Actually generates code.

AHO-JOHNSON ALGORITHM: COVER

- An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.

AHO-JOHNSON ALGORITHM: COVER

- An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.
- The algorithm which decides whether an instruction covers a node also provides a related information

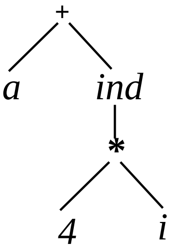
AHO-JOHNSON ALGORITHM: COVER

- An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.
- The algorithm which decides whether an instruction covers a node also provides a related information
  - which of the subtrees of the node should be evaluated in registers (regset)

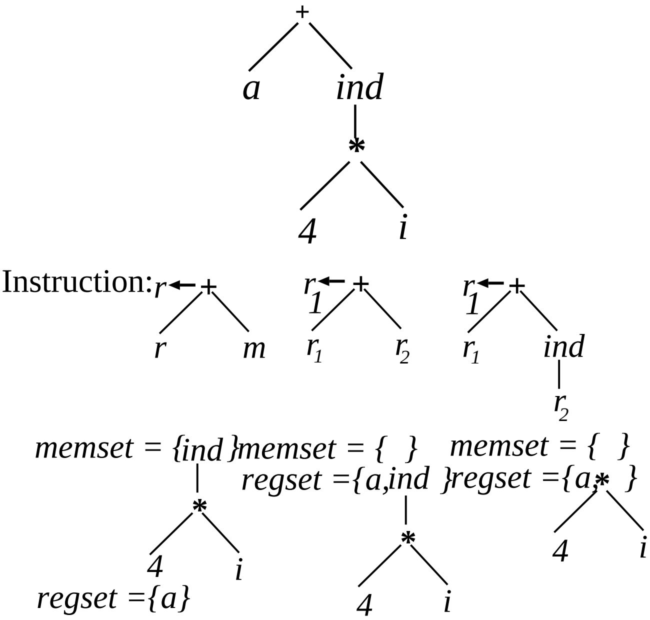
AHO-JOHNSON ALGORITHM: COVER

- An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.
- The algorithm which decides whether an instruction covers a node also provides a related information
  - which of the subtrees of the node should be evaluated in registers (regset)
  - which should be evaluated in memory locations (memset).

EXAMPLE



EXAMPLE


+
a
ind
*
4
i
Instruction:
r ← +
r
m
r1 ← +
r1
r2
r1 ← +
r1
ind
r2
memset = {ind }
*
4
i
regset ={a}
memset = { }
regset ={a,ind }
*
4
i
memset = { }
regset ={a,* }
4
i

ALGORITHM FOR COVER

function *cover*(E, S);

(* decides whether $z \leftarrow E$ covers the expression tree S. If so, then *regset* and *memset* will contain the subtrees of S to be evaluated in register and memory *)

ALGORITHM FOR COVER

function *cover*(E, S);

(* decides whether $z \leftarrow E$ covers the expression tree S. If so, then *regset* and *memset* will contain the subtrees of S to be evaluated in register and memory *)

1. If E is a single register node, add S to *regset* and return *true*.

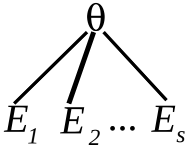
ALGORITHM FOR COVER

function *cover*(E, S);
(* decides whether $z \leftarrow E$ covers the expression tree S. If so, then *regset* and *memset* will contain the subtrees of S to be evaluated in register and memory *)

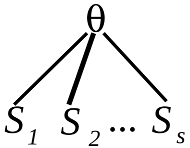
1. If E is a single register node, add S to *regset* and return *true*.
2. If E is a single memory node, add S to *memset* and return *true*.

ALGORITHM FOR COVER

3. If E has the form



then, if the root of S is not θ, return *false*. Else, write S as



For all i from 1 to s do cover(E_i,S_i). Return *true*, only if all invocations return *true*.

AHO-JOHNSON ALGORITHM

Calculates an array of costs $C_j(S)$ for every subtree S of T, whose meaning is to be interpreted as follows:

AHO-JOHNSON ALGORITHM

Calculates an array of costs $C_j(S)$ for every subtree S of T, whose meaning is to be interpreted as follows:

- $C_0(S)$: cost of evaluating S in a memory location.

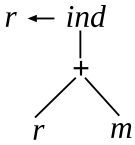
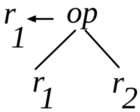
AHO-JOHNSON ALGORITHM

Calculates an array of costs $C_j(S)$ for every subtree S of T, whose meaning is to be interpreted as follows:

- $C_0(S)$: cost of evaluating S in a memory location.
- $C_j(S), j \neq 0$ is the minimum cost of evaluating S using j registers.

EXAMPLE

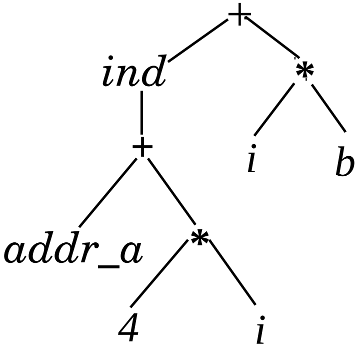
Consider a machine with the instructions shown below.

$r \leftarrow c$	**{MOV #c, r}**
$r \leftarrow m$	**{MOV m, r}**
$m \leftarrow r$	**{MOV r, m}**
	**{MOV m(r), r}**
	**{op r_2, r_1}**

Note that there are no instructions of the form *op m, r* OR *op r, m*.

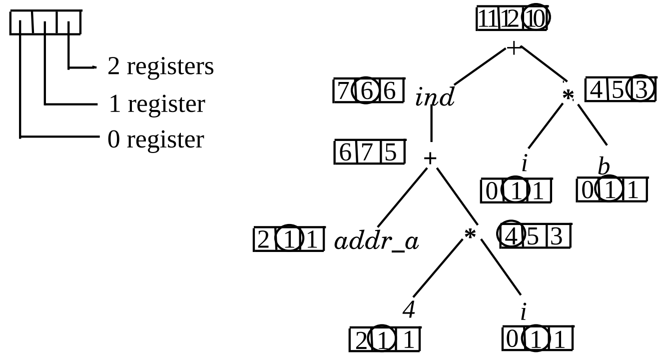
AHO-JOHNSON ALGORITHM

Cost computation with 2 registers for the expression tree

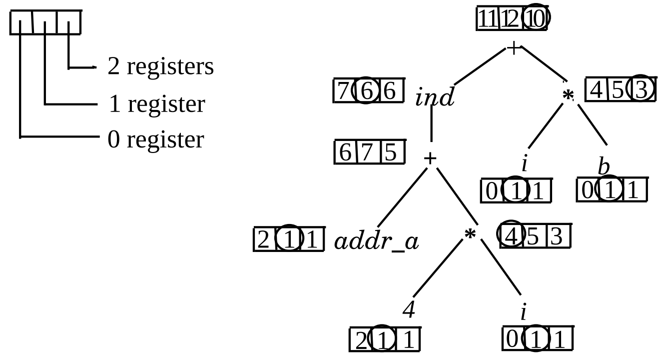


Assume that 4, being a literal, does not reside in memory.

AHO-JOHNSON ALGORITHM



AHO-JOHNSON ALGORITHM



In this example, we assume that 4, being a literal, does not reside in memory. The circles around the costs indicate the choices at the children which resulted in the circled cost of the parent. The next slide explains how to calculate the cost at each node.

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.
2. $C[2] = 1$, the extra register does not help.

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.
2. $C[2] = 1$, the extra register does not help.
3. $C[0] = 2$, load into a register, and then store in memory location.

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.
2. $C[2] = 1$, the extra register does not help.
3. $C[0] = 2$, load into a register, and then store in memory location.

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.
2. $C[2] = 1$, the extra register does not help.
3. $C[0] = 2$, load into a register, and then store in memory location.

For the leaf labeled i,

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.
2. $C[2] = 1$, the extra register does not help.
3. $C[0] = 2$, load into a register, and then store in memory location.

For the leaf labeled i,

1. $C[1] = 1$, load the variable into a register.

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.
2. $C[2] = 1$, the extra register does not help.
3. $C[0] = 2$, load into a register, and then store in memory location.

For the leaf labeled i,

1. $C[1] = 1$, load the variable into a register.
2. $C[2] = 1$,

AHO-JOHNSON ALGORITHM

Consider the subtree $4 * i$. For the leaf labeled 4,

1. $C[1] = 1$, load the constant into a register using the MOVE c, m instruction.
2. $C[2] = 1$, the extra register does not help.
3. $C[0] = 2$, load into a register, and then store in memory location.

For the leaf labeled i,

1. $C[1] = 1$, load the variable into a register.
2. $C[2] = 1$,
3. $C[0] = 0$, do nothing, i is already in a memory location.

AHO-JOHNSON ALGORITHM

For the node labeled *,

AHO-JOHNSON ALGORITHM

For the node labeled *,

1. $C[2] = 3$, evaluate each of the operands in registers and use the op r_1, r_2 instruction.

AHO-JOHNSON ALGORITHM

For the node labeled *,

1. $C[2] = 3$, evaluate each of the operands in registers and use the op r_1, r_2 instruction.
2. $C[0] = 4$, evaluate the node using two registers as above and store in a memory location.

AHO-JOHNSON ALGORITHM

For the node labeled *,

1. $C[2] = 3$, evaluate each of the operands in registers and use the op r_1, r_2 instruction.
2. $C[0] = 4$, evaluate the node using two registers as above and store in a memory location.
3. $C[1] =$

AHO-JOHNSON ALGORITHM

For the node labeled *,

1. $C[2] = 3$, evaluate each of the operands in registers and use the op r_1, r_2 instruction.
2. $C[0] = 4$, evaluate the node using two registers as above and store in a memory location.
3. $C[1] =$

AHO-JOHNSON ALGORITHM

For the node labeled *,

1. $C[2] = 3$, evaluate each of the operands in registers and use the op r_1, r_2 instruction.
2. $C[0] = 4$, evaluate the node using two registers as above and store in a memory location.
3. $C[1] = 5$, notice that our machine has no op m, r instruction. So we can use two registers to perform the operation and store the result in a memory location releasing the registers. When we want to use the result, we can load it in a register. The cost in this case is $C[0] + 1 = 5$.

AHO-JOHNSON ALGORITHM

0. Let n denote the max number of available registers. Set $C_j(s) = \infty$ for all subtrees S of T and for all j, $0 \leq j \leq n$. Visit the tree in postorder. For each node S in the tree do steps 1–3.

AHO-JOHNSON ALGORITHM

0. Let n denote the max number of available registers. Set $C_j(s) = \infty$ for all subtrees S of T and for all j, $0 \leq j \leq n$. Visit the tree in postorder. For each node S in the tree do steps 1–3.
1. If S is a leaf (variable), set $C_0(S) = 0$.

AHO-JOHNSON ALGORITHM

0. Let n denote the max number of available registers. Set $C_j(s) = \infty$ for all subtrees S of T and for all j, $0 \leq j \leq n$. Visit the tree in postorder. For each node S in the tree do steps 1–3.
1. If S is a leaf (variable), set $C_0(S) = 0$.
2. Consider each instruction $r \leftarrow E$ which covers S. For each instruction obtain the *regset* $\{S_1, \ldots, S_k\}$ and *memset* $\{T_1, \ldots, T_l\}$. Then for each permutation π of $\{1, \ldots, k\}$ and for all j, $k \leq j \leq n$, compute

   $$C_j(S) = min(C_j(S), \Sigma_{i=1}^{k} C_{j-i+1}(S_{\pi(i)}) + \Sigma_{i=1}^{l} C_0(T_i) + 1)$$

   Remember the π that gives minimum $C_j(S)$.

AHO-JOHNSON ALGORITHM

0. Let n denote the max number of available registers. Set $C_j(s) = \infty$ for all subtrees S of T and for all j, $0 \le j \le n$. Visit the tree in postorder. For each node S in the tree do steps 1–3.
1. If S is a leaf (variable), set $C_0(S) = 0$.
2. Consider each instruction $r \leftarrow E$ which covers S. For each instruction obtain the *regset* $\{S_1, \ldots, S_k\}$ and *memset* $\{T_1, \ldots, T_l\}$. Then for each permutation π of $\{1, \ldots, k\}$ and for all j, $k \le j \le n$, compute

   $$C_j(S) = min(C_j(S), \Sigma_{i=1}^{k} C_{j-i+1}(S_{\pi(i)}) + \Sigma_{i=1}^{l} C_0(T_i) + 1)$$

   Remember the π that gives minimum $C_j(S)$.
3. Set $C_0(S) = min(C_0(S), C_n(S) + 1)$, and $C_j(S) = min(C_j(S), C_0(S) + 1)$.

AHO-JOHNSON ALGORITHM: NOTES

1. In step 2,
   - $\Sigma_{i=1}^{k} C_{j-i+1}(S_{\pi(i)})$ is the cost of computing the subtrees S_i in registers,
   - $\Sigma_{i=1}^{l} C_0(T_i)$ is the cost of computing the subtrees T_i in memory,
   - 1 is the cost of the instruction at the root.
2. $C_0(S) = min(C_0(S), C_n(S) + 1)$ is the cost of evaluating a node in memory location by first using n registers and then storing it.

AHO-JOHNSON ALGORITHM: NOTES

3. $C_j(S) = min(C_j(S), C_0(S) + 1)$ is the cost of evaluating a node by first evaluating it in a memory location and then loading it.
4. The algorithm also records at each node, the minimum cost, and
   a. The instruction which resulted in the minimum cost.
   b. The permutation which resulted in the minimum cost.

AHO-JOHNSON ALGORITHM: PASS2

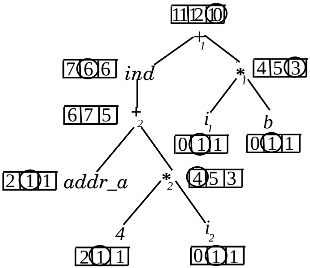
- This pass marks the nodes which have to be evaluated into memory.
- The algorithm is initially invoked as $mark(T, n)$, where T is the given expression tree and n the number of registers supported by the machine.
- It returns a sequence of nodes $x_1, \ldots, x_{s-1}$, where $x_1, \ldots, x_{s-1}$ represent the nodes to be evaluated in memory. For purely technical reasons, after *mark* returns, x_s is set to T itself.

function *mark*(S, j)

1. Let $z \leftarrow E$ be the optimal instruction associated with $C_j(S)$, and π be the optimal permutation. Invoke *cover*(E, S) to obtain regset $\{S_1, \ldots, S_k\}$ and memset $\{T_1, \ldots, T_l\}$ of S.
2. For all i from 1 to k do *mark*$(S_{\pi(i)}, j - i + 1)$.
3. For all i from 1 to l do *mark*(T_i, n).
4. If j is n and the instruction $z \leftarrow E$ is a store, increment s and set x_s to the root of S.
5. Return.

AHO-JOHNSON ALGORITHM

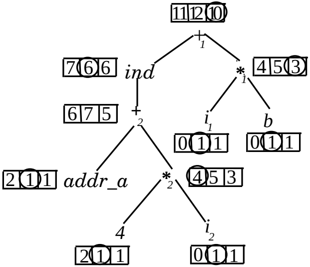
$$mark(+_1, 2)$$



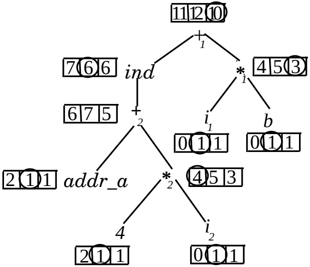
AHO-JOHNSON ALGORITHM

$mark(+_1, 2)$

$mark(*_1, 2)$



AHO-JOHNSON ALGORITHM

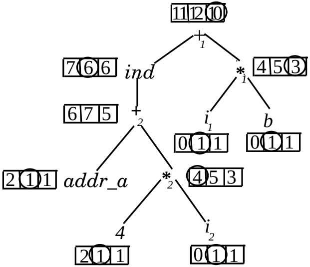


$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

AHO-JOHNSON ALGORITHM



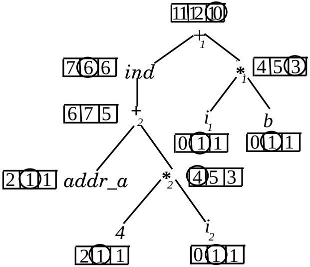
$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

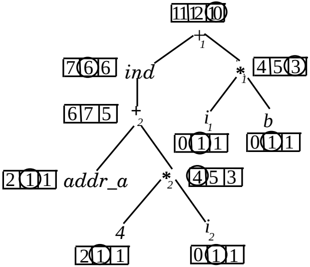
$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

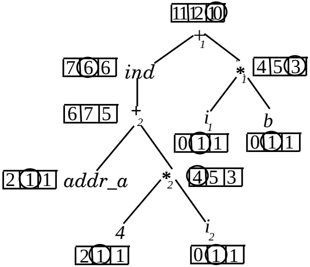
$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

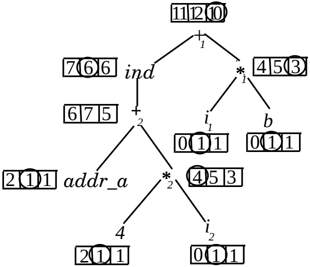
$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

$mark(addr_a, 1)$

AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$\quad mark(*_1, 2)$

$\quad\quad mark(i_1, 2)$

$\quad\quad mark(b_1, 1)$

$\quad mark(ind, 1)$

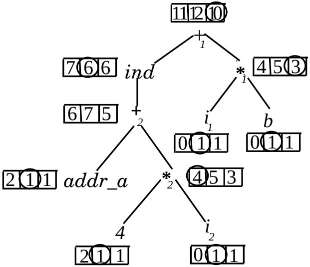
$\quad\quad mark(+_2, 1)$

$\quad\quad\quad mark(addr_a, 1)$

$\quad\quad\quad mark(*_2, 2)$ //the covering

//instruction is $m \leftarrow \ldots$

AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

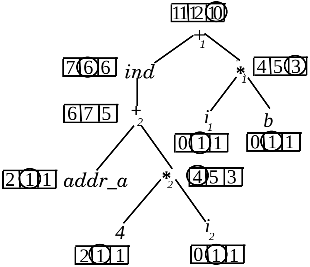
$mark(addr_a, 1)$

$mark(*_2, 2)$ //the covering

//instruction is $m \leftarrow \ldots$

$mark(4, 2)$

AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

$mark(addr_a, 1)$

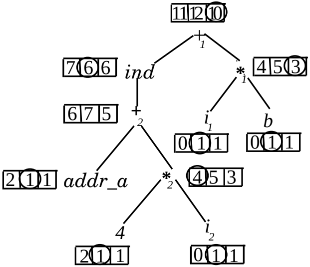
$mark(*_2, 2)$ //the covering

//instruction is $m \leftarrow \ldots$

$mark(4, 2)$

$mark(i_2, 1)$

AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

$mark(addr_a, 1)$

$mark(*_2, 2)$ //the covering

//instruction is $m \leftarrow \ldots$

$mark(4, 2)$

$mark(i_2, 1)$

$x_1 = *_2$ // $*_2$ needs to be stored

AHO-JOHNSON ALGORITHM: PASS 3

- The algorithm generates code for the subtrees rooted at $x_1, \ldots x_s$, in that order.

AHO-JOHNSON ALGORITHM: PASS 3

- The algorithm generates code for the subtrees rooted at $x_1, \ldots x_s$, in that order.
- After generating code for x_i, the algorithm replaces the node with a distinct memory location m_i.

AHO-JOHNSON ALGORITHM: PASS 3

- The algorithm generates code for the subtrees rooted at $x_1, \ldots x_s$, in that order.
- After generating code for x_i, the algorithm replaces the node with a distinct memory location m_i.
- The algorithm uses the following unspecified routines

AHO-JOHNSON ALGORITHM: PASS 3

- The algorithm generates code for the subtrees rooted at $x_1, \ldots x_s$, in that order.
- After generating code for x_i, the algorithm replaces the node with a distinct memory location m_i.
- The algorithm uses the following unspecified routines
  - *alloc* {*allocates a register*}

AHO-JOHNSON ALGORITHM: PASS 3

- The algorithm generates code for the subtrees rooted at $x_1, \ldots x_s$, in that order.
- After generating code for x_i, the algorithm replaces the node with a distinct memory location m_i.
- The algorithm uses the following unspecified routines
  - *alloc* {*allocates a register*}
  - *free* {*frees a register*}

AHO-JOHNSON ALGORITHM

The main program is:

1. Set $i = 1$ and invoke $\mathit{code}(x_i, n)$. Let α be the register returned. Issue the instruction $m_i \leftarrow \alpha$, invoke $\mathit{free}(\alpha)$, and rewrite x_i to represent m_i. Repeat this step for $i = 2, \ldots, s - 1$.

AHO-JOHNSON ALGORITHM

The main program is:

1. Set $i = 1$ and invoke $code(x_i, n)$. Let α be the register returned. Issue the instruction $m_i \leftarrow \alpha$, invoke $free(\alpha)$, and rewrite x_i to represent m_i. Repeat this step for $i = 2, \ldots, s-1$.
2. Invoke $code(x_s, n)$.

AHO-JOHNSON ALGORITHM

The main program is:

1. Set $i = 1$ and invoke $code(x_i, n)$. Let α be the register returned. Issue the instruction $m_i \leftarrow \alpha$, invoke $free(\alpha)$, and rewrite x_i to represent m_i. Repeat this step for $i = 2, \ldots, s-1$.
2. Invoke $code(x_s, n)$.

AHO-JOHNSON ALGORITHM

The main program is:

1. Set $i = 1$ and invoke $code(x_i, n)$. Let α be the register returned. Issue the instruction $m_i \leftarrow \alpha$, invoke $free(\alpha)$, and rewrite x_i to represent m_i. Repeat this step for $i = 2, \ldots, s-1$.
2. Invoke $code(x_s, n)$.

This uses the function $code(S, j)$ which generates code for the tree S using j registers, and also returns the register in which the code was evaluated. This is described in the following slide.

function $code(S, j)$

1. Let $z \leftarrow E$ be the optimal instruction for $C_j(S)$, and π be the optimal permutation. Invoke $cover(E, S)$ to obtain the regset $\{S_1, \ldots, S_k\}$.

function $code(S, j)$

1. Let $z \leftarrow E$ be the optimal instruction for $C_j(S)$, and π be the optimal permutation. Invoke $cover(E, S)$ to obtain the regset $\{S_1, \ldots, S_k\}$.
2. For $i = 1$ to k, do $code(S_{\pi(i)}, j - i + 1)$. Let $\alpha_1, \ldots, \alpha_k$ be the registers returned.

function $code(S, j)$

1. Let $z \leftarrow E$ be the optimal instruction for $C_j(S)$, and π be the optimal permutation. Invoke $cover(E, S)$ to obtain the regset $\{S_1, \ldots, S_k\}$.
2. For $i = 1$ to k, do $code(S_{\pi(i)}, j - i + 1)$. Let $\alpha_1, \ldots, \alpha_k$ be the registers returned.
3. If $k = 0$, call alloc to obtain an unused register to return.

function $code(S, j)$

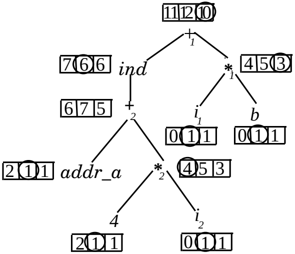
1. Let $z \leftarrow E$ be the optimal instruction for $C_j(S)$, and π be the optimal permutation. Invoke $cover(E, S)$ to obtain the regset $\{S_1, \ldots, S_k\}$.
2. For $i = 1$ to k, do $code(S_{\pi(i)}, j - i + 1)$. Let $\alpha_1, \ldots, \alpha_k$ be the registers returned.
3. If $k = 0$, call alloc to obtain an unused register to return.
4. Issue $\alpha \leftarrow E$ with $\alpha_1, \ldots \alpha_k$ substituted for the registers of E. Memory locations of E are substituted by some m_i or leaves of T.

function *code*(S, j)

1. Let $z \leftarrow E$ be the optimal instruction for $C_j(S)$, and π be the optimal permutation. Invoke *cover*(E, S) to obtain the regset $\{S_1, \dots, S_k\}$.
2. For $i = 1$ to k, do *code*$(S_{\pi(i)}, j - i + 1)$. Let $\alpha_1, \dots, \alpha_k$ be the registers returned.
3. If $k = 0$, call alloc to obtain an unused register to return.
4. Issue $\alpha \leftarrow E$ with $\alpha_1, \dots \alpha_k$ substituted for the registers of E. Memory locations of E are substituted by some m_i or leaves of T.
5. Call *free* on $\alpha_1, \dots \alpha_k$ except α. Return α as the register for *code*(S, j).

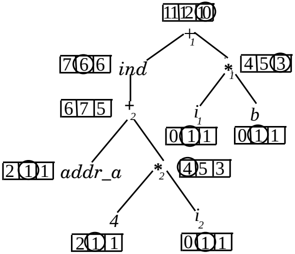
AHO-JOHNSON ALGORITHM

EXAMPLE: For the expression tree shown below, the code generated will be:



AHO-JOHNSON ALGORITHM

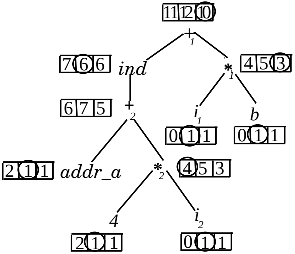
EXAMPLE: For the expression tree shown below, the code generated will be:



MOVE #4, r_1 (evaluate $4 * i$ first, since
MOVE i, r_2 this node has to be stored)
MUL r_2, r_1

AHO-JOHNSON ALGORITHM

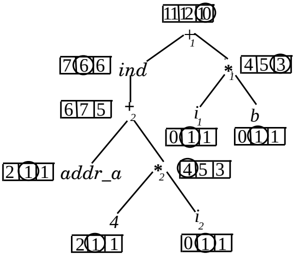
EXAMPLE: For the expression tree shown below, the code generated will be:



MOVE #4, r_1 (evaluate $4 * i$ first, since
MOVE i, r_2 this node has to be stored)
MUL r_2, r_1
MOVE r_1, m_1

AHO-JOHNSON ALGORITHM

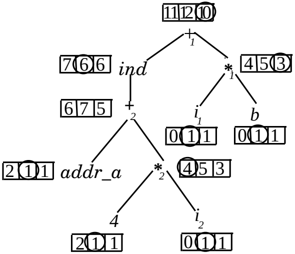
EXAMPLE: For the expression tree shown below, the code generated will be:



MOVE $\#4,\ r_1$ (evaluate $4 * i$ first, since
MOVE $i,\ r_2$ this node has to be stored)
MUL $r_2,\ r_1$
MOVE $r_1,\ m_1$
MOVE $i,\ r_1$ (evaluate $i * b$ next, since this
MOVE $b,\ r_2$ requires 2 registers)
MUL $r_2,\ r_1$

AHO-JOHNSON ALGORITHM

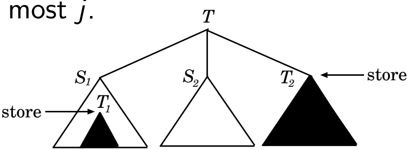
EXAMPLE: For the expression tree shown below, the code generated will be:



$MOVE\ \#4,\ r_1$ (evaluate $4 * i$ first, since
$MOVE\ i,\ r_2$ this node has to be stored)
$MUL\ r_2,\ r_1$
$MOVE\ r_1,\ m_1$
$MOVE\ i,\ r_1$ (evaluate $i * b$ next, since this
$MOVE\ b,\ r_2$ requires 2 registers)
$MUL\ r_2,\ r_1$
$MOVE\ \#addr\_a,\ r_1$
$MOVE\ m_1(r_1),\ r_1$ (evaluate the *ind* node)
$ADD\ r_1,\ r_2$ (evaluate the root)

PROOF OF OPTIMALITY

THEOREM: $C_j(T)$ is the minimal cost over all strong normal form programs $P_1 J_1 \ldots P_{s-1} J_{s-1} P_s$ which compute T such that the width of P_s is at most j.



- Consider an optimal program $P_1 J_1 P_2 J_2 P I$ in strong normal form.
- Now P is a strongly contiguous program which evaluates in registers values required by I. So P might be written as a sequence of contiguous programs, say $P_3 P_4$.
- For instance, P_3 could be the program computing the portion of S_1 in figure the figure which is not shaded, using j registers, and P_4 could be computing S_2 using $j - 1$ registers. Also $P_1 J_1$ and $P_2 J_2$ must be computing the shaded subtrees T_1 and T_2.

AHO-JOHNSON ALGORITHM

Now let us calculate the cost of this program.

- $P_1 J_1 P_3$ is a program in strong normal form, evaluating the subtree S_1. Since the width of P_3 is j, as induction hypothesis we can assume that the cost of $P_1 J_1 P_3$ is atleast $C_j(S_1)$.
- P_4 is also a program in strong normal form, evaluating S_2 and the width of P_4 is $j-1$. Once again, as induction hypothesis, we can assume that the cost of P_4 is atleast $C_{j-1}(S_2)$.
- Finally $P_2 J_2$ is a program which computes the subtree T_2 and stores it in memory. The cost of this is no more than $C_0(T_2)$.

Therefore the cost of this optimal program is $1 + C_j(S_1) + C_{j-1}(S_2) + C_0(T_2)$. The program generated by our algorithm is no costlier than this (Pass 1, step 2), and is therefore optimal.

AHO-JOHNSON ALGORITHM

COMPLEXITY OF THE ALGORITHM

1. The time required by Pass 1 is an, where a is a constant depending
    - linearly on the size of the instruction set
    - exponentially on the arity of the machine, and
    - linearly on the number of registers in the machine

   and n is the number of nodes in the expression tree.
2. Time required by Passes 2 and 3 is proportional to n

Therefore the complexity of the algorithm is $O(n)$.