CODE GENERATION

TEACHING MATERIAL

1. **Instructor's Slide-set** ... **1**
   (Inclusive of worked examples)

2. **Graded exercises** ... **86**

CODE GENERATION

In this module, we discuss code generation techniques for expressions. The primary concern here is the efficient utilisation of machine resources, viz. machine instructions and registers.

We present the material in two steps :

- A simple code generator. This discusses simple techniques and data structures for performing instruction selection and management of machine registers.
- Algorithms for making efficient utilisation of machine resources.

Note :

The module on Static Semantic analysis (module 5) discusses code generation for control structures of the language.

A SIMPLE CODE GENERATOR

1. Assumptions made in this material

   (a) Target machine is as defined in section 9.2 of the book by Aho, Sethi and Ullman, viz.

   - all instructions are two-operand instructions
   - the machine has n registers named R0, R1 $\cdots$ R$n-1$.

   (b) All operators in an expression are commutative.

   (c) The code generator is driven by a syntax/semantic analyser using Syntax Directed Translation (SDT). Hence, the code generator can be used in a single-pass compiler.

Each of these assumptions can be relaxed through appropriate generalisations.

A SIMPLE CODE GENERATOR

2. Operand Descriptors

All details required for addressing an operand are stored in an *Operand Descriptor*.

The operands encountered during code generation may reside in different *address spaces* (e.g. local and non-local address spaces). They might also reside in different elements of the machine hardware during execution of the generated code.

Both these factors influence *how* the operand should be accessed in a machine instruction.

For example,

(a) An operand may reside in a machine register (e.g. it may be the result of a previous operation)

(b) An operand may reside in a memory location which can be accessed using an absolute address or base-displacement addressing (e.g. a *local* operand)

(c) An operand may reside in a memory location which may require indirect addressing (e.g. a *non-local* operand)

We define *operand addressibility modes* to indicate how an operand should be accessed.

A SIMPLE CODE GENERATOR

2. Operand Descriptors (Contd.)

The code to be generated for an expression depends on the addressibilities of its operands.

Example

$$a := b + c;$$

1. b and c are in memory :

```
MOV     B, R0
ADD     C, R0
MOV     R0, A
```

2. c is in register R0 (***note :*** '+' is commutative) :

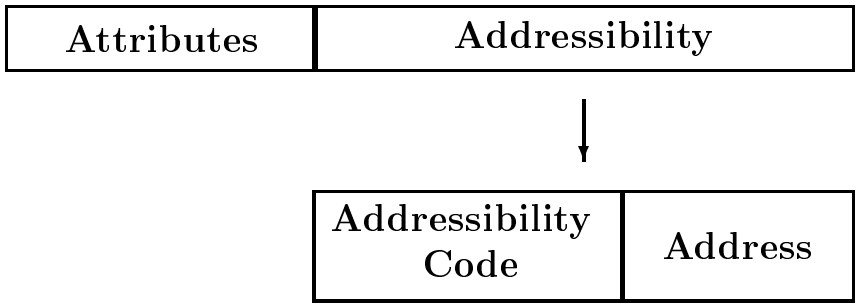
```
ADD     B, R0
MOV     R0, A
```

3. b is a local variable and c is non-local :

   This is left as an exercise for the students.

A SIMPLE CODE GENERATOR

2. Operand Descriptors (Contd.)



Typical addressibility codes are —

(a) 'S' : Value in Storage

(b) 'R' : Value in Register

(c) 'AS' : Address in Storage

(d) 'AR' : Address in Register

Notes:

1. Attributes could be type, size, etc.
2. The 'address' field contains the name of a register or the address of a storage location. The storage address may have components like (base, displacement) or (Display index, displacement in activation record), etc.

A SIMPLE CODE GENERATOR

2. Operand Descriptors (Contd.)

Question : When do addressibility codes AS and AR arise ?

1. **Addressibility AS (address in storage) arises for parameters of a procedure call. A list of addresses of the actual parameters is typically constructed in memory (possibly in an activation record).**
2. **Addressibility AR (address in register) typically arises for references to array elements. First we need to perform *address arithmetic* to compute the address of the element to be referenced. This leaves the effective address of the element in a register.**

   Example : $x := y + info[i, j]$;

A SIMPLE CODE GENERATOR

2. Operand Descriptors (Contd.)

Constructing Operand Descriptors

(a) An operand descriptor is constructed when an entity is identified as a potential operand.

(b) The descriptor may be constructed in a table (viz. Symbol Table entry of a variable) or elsewhere.

(c) For simplicity we will assume operand descriptors to be stored in an array called ***Operand_descr***.

Example

$$a * b + c \cdots$$

1. Build descriptors for a and b in $operand_descr[1]$ and $operand_descr[2]$ respectively.
2. Decide on the code to be generated.
3. Build a descriptor for the result (since it is an operand for the subsequent evaluation of '+').

A SIMPLE CODE GENERATOR

3. Code Generator Schematic

Initially we assume the use of a single register.

Exp	:	$Exp + Term$	**{$$ = codegen(+, $1, $3)}**
	|	$Term$	**{$$ = $1}**
	;		
$Term$	:	$Term * Factor$	**{$$ = codegen(*, $1, $3)}**
	|	$Factor$	**{$$ = $1}**
	;		
$Factor$	:	id	**{$$ = build_descriptor($1)}**
	;		

%%

```
build_descriptor  (opd1);
 {
     i = i + 1;
     operand_descr[i] = (type, addressibility, address) of opd1;
     return i;
 }

codegen  (operator, opd1, opd2);
 {
     /* Details on next page */
 }
```

A SIMPLE CODE GENERATOR

3. Code Generator Schematic (Contd.)

```
codegen (operator, opd1, opd2);
{
    if opd1.addressibility = 'R' then
            /* Code generation – case 1 */
        if operator = '+' generate ( 'ADD opd2, R0' );
        else if operator = '*' then ···
    else if opd2.addressibility = 'R' then
            /* Code generation – case 2 */
        if operator = '+' generate ( 'ADD opd1, R0' );
        else ···
    else
    {
            /* Code generation – case 3 */
        generate ('MOV opd1, R0');
        if operator = '+' generate ( 'ADD opd2, R0' );
        else ···
    }
            /* Common part – Create a new descriptor
                Saying operand value is in register R0 */
    i = i + 1;
    operand_descr[i] = (< type >, 'R', R0);
    return i;
}
```

A SIMPLE CODE GENERATOR

Example

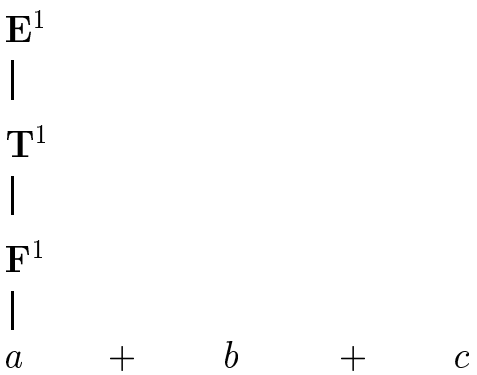
Code generation for the expression

$$a \ + \ b \ + \ c$$

Step 1 :

Parsing actions		***Operand_Descr*** ($i = 0$ **initially**)		
$\mathbf{F}^1 \leftarrow$ **a**	**1**	**int**	**S**	**Ad(a)**
$\mathbf{T}^1 \leftarrow \mathbf{F}^1$		– **Same** –		
$\mathbf{E}^1 \leftarrow \mathbf{T}^1$		– **Same** –		

Parse tree :



A SIMPLE CODE GENERATOR

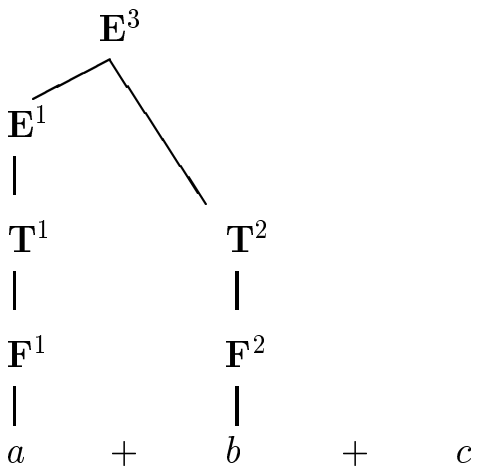
Example (Contd.) : Code generation for the expression

$$a \ + \ b \ + \ c$$

Step 2 :

Parsing actions		***Operand_Descr***		
	1	int	S	Ad(a)
$\mathbf{F^2 \leftarrow b}$	**2**	int	S	Ad(b)
$\mathbf{T^2 \leftarrow F^2}$		– Same –		
	1	int	S	Ad(a)
	2	int	S	Ad(b)
$\mathbf{E^3 \leftarrow E^1 + T^2}$	**3**	int	R	R0

Parse tree :



Generated code :

```
MOV     A, R0
ADD     B, R0
```

A SIMPLE CODE GENERATOR

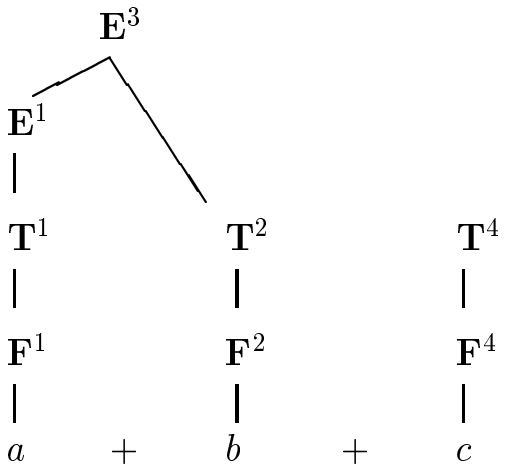
Example (Contd.) : Code generation for the expression

$$a \ + \ b \ + \ c$$

Step 3 :

Parsing actions		***Operand_Descr***		
	1	**int**	**S**	**Ad(a)**
	2	**int**	**S**	**Ad(b)**
	3	**int**	**R**	**R0**
$\mathbf{F}^4 \leftarrow \mathbf{c}$	**4**	**int**	**S**	**Ad(c)**
$\mathbf{T}^4 \leftarrow \mathbf{F}^4$		**– Same –**		

Parse tree :



A SIMPLE CODE GENERATOR

Example (Contd.) : Code generation for the expression

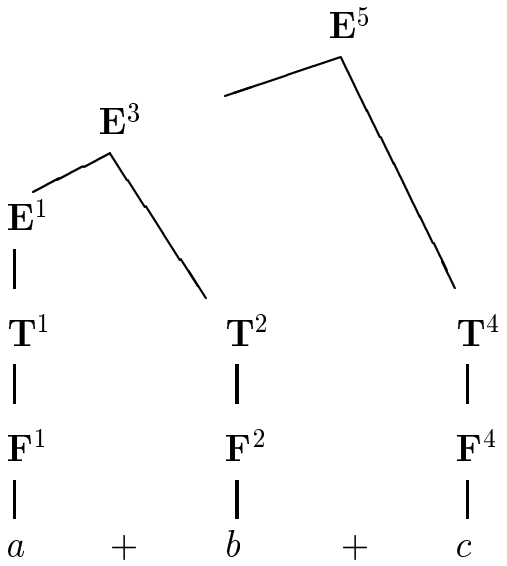
$$a \ + \ b \ + \ c$$

Step 4 :

Parsing actions ***Operand_Descr***

1	**int**	**S**	**Ad(a)**
2	**int**	**S**	**Ad(b)**
3	**int**	**R**	**R0**
4	**int**	**S**	**Ad(c)**
$\mathbf{E}^5 \leftarrow \mathbf{E}^3 + \mathbf{T}^4$ **5**	**int**	**R**	**R0**

Parse tree :



Generated code : **ADD C, R0**

A SIMPLE CODE GENERATOR

4. Handling Partial Results

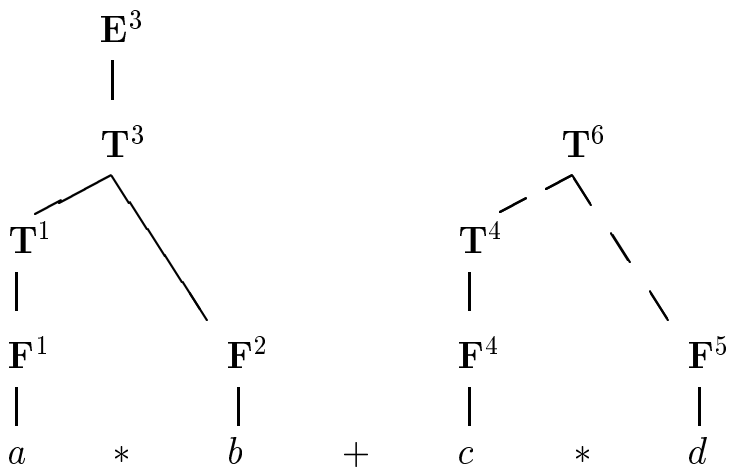
Example : Code generation for the expression

$$a * b + c * d$$

Situation prior to code generation for $c * d$:

Parsing actions		***Operand_Descr***			
$\mathbf{F}^4 \leftarrow c$	**1**	**int**	**S**	**Ad(a)**	
	2	**int**	**S**	**Ad(b)**	
$\mathbf{T}^4 \leftarrow \mathbf{F}^4$	**3**	**int**	**R**	**R0**	$\leftarrow a * b$
	4	**int**	**S**	**Ad(c)**	
$\mathbf{F}^5 \leftarrow d$	**5**	**int**	**S**	**Ad(d)**	

Parse tree :



Generated code :

```
MOV     A, R0
MUL     B, R0
```

A SIMPLE CODE GENERATOR

4. Handling Partial Results

Example (Contd.) : Code generation for the expression

$$a * b + c * d$$

Generated code :

```
MOV     A, R0
MUL     B, R0
```

Code about to be generated :

```
MOV     C, R0
MUL     D, R0
```

This code would destroy the contents of register R0 !

Hence partial results need to be saved into a memory location at an appropriate point to free a register.

Q : **When should the register contents be saved ?**

A SIMPLE CODE GENERATOR

4. Handling Partial Results

Questions :

When should the contents of a register be saved ?

1. **A naive approach : Save the contents of the register after every operation !**

   Example : $a + b + c$

MOV	**A, R0**
ADD	**B, R0**
MOV	**R0, Temp$_1$**
MOV	**Temp$_1$, R0**
ADD	**C, R0**

2. **Save the contents whenever a low priority operator is met ...**

   Example : $a * b + c * \cdots$

   (***Note :*** **This is hard to generalise for** > 1 **registers.**)

3. **...**

A SIMPLE CODE GENERATOR

5. Register Descriptors

Occupied ?	Descriptor No.

(a) The *Occupied ?* field indicates whether the register currently contains a useful result.

(b) If the register is occupied, the *Descriptor number* field points to the operand descriptor of the value currently contained in the register.

(c) Register descriptors are stored in an array called *Register_Descr*.

(d) The *Register_Descr* array contains as many elements as the number of registers in the machine.

A SIMPLE CODE GENERATOR

5. Register Descriptors (Contd.)

Example : Code generation for the expression

$$a * b + c * d$$

Situation after code generation for $a * b$ **:**

Operand_Descr

1	int	S	Ad(a)
2	int	S	Ad(b)
3	int	R	R0

Register_Descr

R0	Occupied	**3**

The register descriptor indicates the present status of the register and fully describes the value contained in it.

A SIMPLE CODE GENERATOR

6. Saving of Partial Results

- ***When*** should a partial result, computed in a machine register during the evaluation of an expression, be saved ?

  (a) When an independent evaluation is starting (i.e. the partial result is ***not*** an operand of the next evaluation), and

  (b) All registers are occupied.

- ***Where*** should a partial result be saved ?

  We can assume a set of ***temporary locations*** named TEMP_1, TEMP_2 ... to be available for saving partial results. Whenever it becomes necessary to save a partial result into the memory, the compiler can use the *next* temporary location.

A SIMPLE CODE GENERATOR

6. Saving of Partial Results (Contd.)

- ***How*** **to save a Partial Result ?**

  (a) **Perform the following :**

  1. **Generate an instruction**

     MOV ***reg_no***, TEMP_j

     for the selected register ***reg_no***, **and**
  2. **Increment** j **by '1'.**

  (b) **Change the descriptor in**

  Operand_descr(***Register_descr***(***reg_no***).Descriptor no.)

  to

$< type >$	S	Ad(TEMP_j)

  (c) **Set** ***Register_descr***(***reg_no***) = (Not occupied, —).

A SIMPLE CODE GENERATOR

The complete code generation routine :

```
codegen  (operator, opd1, opd2);
 {
     if  opd1.addressibility = 'R'  then
                 /* Code generation – case 1 */
           if  operator = '+'  generate  ( 'ADD opd2, R0' );
           else  ...
     ...
                 /* Code generation – case 3 */                ⇐
     if  Register_descr(R0).Occupied = 'occupied'  then
                 /* First save, if necessary */
                 /* Now generate code */
     {
           generate  ('MOV R0, Temp_j');
           j = j + 1;
           Set Operand_descr(Register_descr(1).Descriptor no)
           to        | < type > | S | Ad(TEMP_j) |
     }
     generate  ('MOV opd1, R0');
     if  operator = '+'  then  generate  ( 'ADD opd2, R0' );
     else  ...
               /* Common part – Create a new descriptor */
     ...
 }
```

A SIMPLE CODE GENERATOR

Example : Code generation for the expression

$$a * b + c * d$$

Step 1 : Situation prior to code generation for $c * d$:

	Operand_Descr			*Register_Descr*	
1	int	S	Ad(a)		
2	int	S	Ad(b)		
3	int	R	R0	Occupied	3
4	int	S	Ad(c)		
5	int	S	Ad(d)		

Generated code : **MOV A, R0**
MUL B, R0

Step 2 : Situation after generating a STORE instruction :

	Operand_Descr			*Register_Descr*	
1	int	S	Ad(a)		
2	int	S	Ad(b)		
3	int	S	Ad($TEMP_1$)	Not Occu.	—
4	int	S	Ad(c)		
5	int	S	Ad(d)		

Generated code : **MOV R0, $TEMP_1$**

A SIMPLE CODE GENERATOR

Example (Contd.)

Step 3 : Situation after code generation for $c * d$:

	Operand_Descr			*Register_Descr*	
1	int	S	Ad(a)		
2	int	S	Ad(b)		
3	int	S	Ad($TEMP_1$)		
4	int	S	Ad(c)	Occupied	6
5	int	S	Ad(d)		
6	int	R	R0		

Generated code : MOV C, R0
MUL D, R0

Step 4 : Situation after code generation for '+' :

	Operand_Descr			*Register_Descr*	
1	int	S	Ad(a)		
2	int	S	Ad(b)		
3	int	S	Ad($TEMP_1$)		
4	int	S	Ad(c)		
5	int	S	Ad(d)	Occupied	7
6	int	R	R0		
7	int	R	R0		

Generated code : ADD $TEMP_1$, R0

A SIMPLE CODE GENERATOR

Assignment :

Augment the code generation schematic presented so far to incorporate the following features :

1. **The parentheses in an expression**
2. **Non-commutative operators like '–' and '/', etc.**

Note :

1. **First augment the syntax of expressions, and the Syntax Directed Translation schematic.**
2. **Modify the action routine 'codegen' to handle noncommutative operators.**

A SIMPLE CODE GENERATOR

7. Managing Temporary Locations

So far, we have assumed that a distinct temporary location is used for saving every partial result. These temporary locations were named TEMP_1, TEMP_2 ... TEMP_j ...

Question :

Is it essential to use a distinct temporary location for each partial result ? In other words, can we minimise the number of temporary locations required ?

Answer :

A temporary location TEMP_j used say, in statement S_i, can surely be re-used in statement S_{i+1} since all statements are independent of one another.

- ***Q :*** Can TEMP_j be re-used within S_i itself ?
- ***A :*** Yes. Temporary locations can be re-used in a ***last-in- first-out*** (LIFO) manner in the unoptimised code for an expression.

A SIMPLE CODE GENERATOR

7. Managing Temporary Locations (Contd.)

Temporary locations can be re-used in a *last-in- first-out* manner in the unoptimised code for an expression.

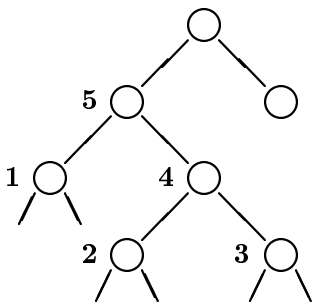
Reasons :

(a) Since common sub-expressions do not exist, a partial result is recalled and used only once, e.g. in

a * b + ...
↑

value of $a * b$ is only used in this addition. Hence, the temporary location containing this value can be released after this use.

(b) expression trees are a natural (LIFO) structure



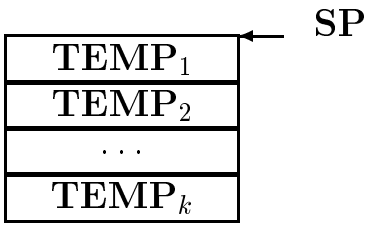
When evaluated left-to-right, value of node 1 would be saved *before* that of node 2 (in order to evaluate node 3), and would be used *after* that of node 2.

A SIMPLE CODE GENERATOR

7. Managing Temporary Locations (Contd.)

Let us assume that we have some k temporary locations. Now,

1. The temporary names $\text{TEMP}_1, \ldots \text{TEMP}_k$ can be put into a stack called *tempstack* at the start of compilation of a statement.
2. Whenever the compiler needs to save a partial result into the memory, it pops a temporary location name off *tempstack*.
3. Whenever the value saved in a temporary location is used in an instruction, the compiler would push the name of the temporary location on *tempstack*.



Question : What is the value of k ?

k is the largest number of temporary locations required by any statement in the program.

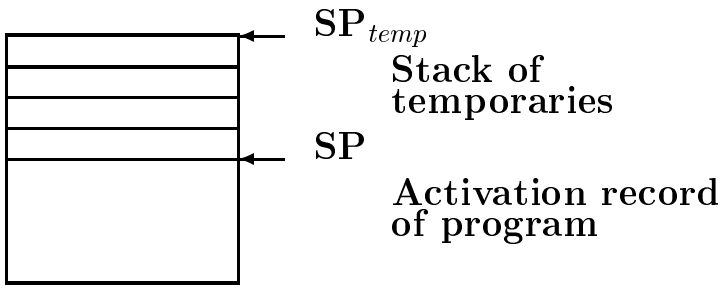
A SIMPLE CODE GENERATOR

7. Managing Temporary Locations (Contd.)

Organising the temporary locations :

1. **Create $\text{TEMP}_1, \ldots \text{TEMP}_k$ as *compiler generated variables*, i.e. compiler generated names for which storage is explicitly allocated in the target program.**

2. **Use an anonymous stack of temporaries *on top* of the activation record of a program.**



Note :

There is no need for distinct names for the temporaries in this case. The compiler will simply use *push* and *pop* actions in the generated code to save and recall partial results. The number of temporaries required (i.e. k) need not be known during compilation.

A SIMPLE CODE GENERATOR

8. Managing register usage for partial results

In un-optimised code for expressions not involving procedure calls :

(i) A partial result should be saved in memory when all registers are occupied and an independent operation is to be evaluated.

(ii) Whenever necessary, save the partial result which is the ***earliest computed result*** still in a register. This ensures the stack behaviour of the temporaries.

Note :

In the simple code generator developed so far, this can be easily achieved using the array of register descriptors.

A SIMPLE CODE GENERATOR

8. Managing register usage for partial results (Contd.)

In optimised code, a partial result may be used many times (e.g. a common sub-expression). Hence,

(a) Consider whether a partial result residing in a register is used more than once in the code. If so, do not destroy it till its last usage point.

(b) Save the partial result (still in register) whose next use is *farthest* in the code. (This performs well in practice, however it is not an optimal strategy.)

Notes :

1. The temporaries can no longer be organised as a stack.
2. We need multiple passes to determine which partial result to save when a register is required.
3. As a consequence of item 2 above, we can no longer use a simple SDT schematic. We must work on the basis of an intermediate code for an expression.

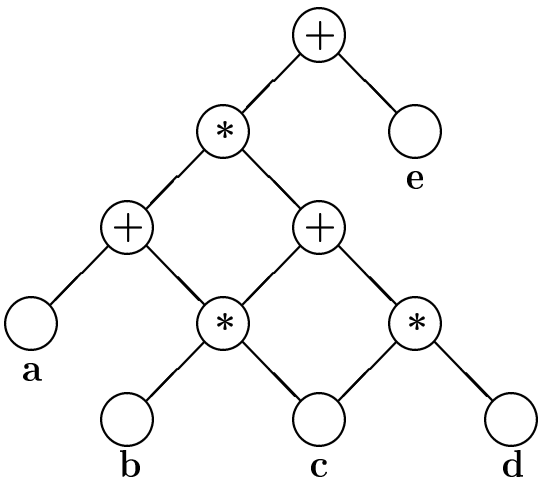
A SIMPLE CODE GENERATOR

9. Practical code generation

In practice, expressions are not always trees, they can be DAG's, i.e. operations can have common nodes.

- A common leaf is a variable/constant used more than once.
- A common internal node represents a shared sub-expression.

Example :



- c is a shared operand,
- $b * c$ is a common sub-expression within this expression.

We will now present an algorithm from A-S-U (page 538).

A SIMPLE CODE GENERATOR

9. Practical code generation (Contd.)

Algorithm from A-S-U (page 538) :

1. **Before loading an operand, check whether its value is already available in a register. For this :**
   - (a) **Keep the descriptor of a variable in the Symbol Table, i.e. a single descriptor / variable.**
   - (b) **For an operand which is a partial result, extend the register descriptor to hold additional information, e.g. last use of the value.**
2. **Delay the Store instructions for assignments, viz.**

   $a := b * c$;

   $x := a$;

   This has two consequences :
   - (a) **Future references to variables a and x can be serviced out of the register. This reduces Load / Store instructions across statements.**
   - (b) **Now, while freeing a register, save its contents into a temporary location (if partial result) and / or all variables whose value it contains (in case of assignments).**

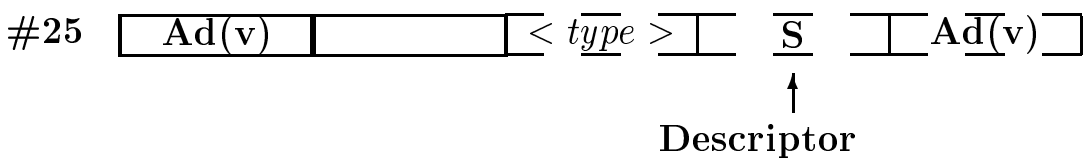
     In above example, the register contains values of expression $b * c$ and variables a and x.

A SIMPLE CODE GENERATOR

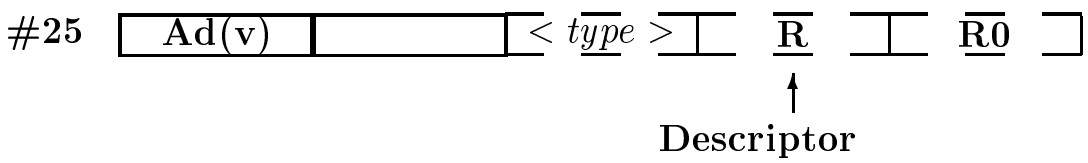
9. Practical Code Generation (Contd.)

Operand Descriptors :

The operand descriptor for variable v is stored in its symbol table entry.



The descriptor is changed to



(a) When v is loaded into a register, or

(b) When code is generated for the assignment

$$v := < exp >;$$

where $< exp >$ is evaluated in register 1.

A SIMPLE CODE GENERATOR

9. Practical Code Generation (Contd.)

Register Descriptors :

Occupied ?	List of Descriptor Nos.

If the register is occupied, the field ***List of Descriptor numbers*** points to the descriptor(s) of the value(s) currently contained in the register.

Implementation notes

1. No operand descriptors need to be built for variables, as their descriptors exist in their symbol table entries.
2. Operand descriptors are built for constants and partial results in the array ***Operand_Descr*** as before.

Salient points of the algorithm are given in the next transparency.

A SIMPLE CODE GENERATOR

9. Practical Code Generation (Contd.)

Elements of the algorithm (Ref. A-S-U pg. 538) :

Function *getreg* determines *where* to evaluate :

$$x \ := \ y \ op \ z \ ;$$

and returns this information in L.

1. If y is in a register which does not hold any other value, and y is not live, then return the address of the register containing y.
2. Failing 1, find an empty register for this use, if one exists.
3. Failing 2, if x has another use in the basic block, or if op is an operation that requires a register (*viz.* indexing), then take an occupied register and save its contents into a temporary location and/or all variable(s) whose value(s) it contains due to assignment(s). Return the address of this register.
4. Failing 3, return the memory location of x as L.

Assignment : Implement this algorithm using the data structures described before.

SETHI-ULLMAN ALGORITHM

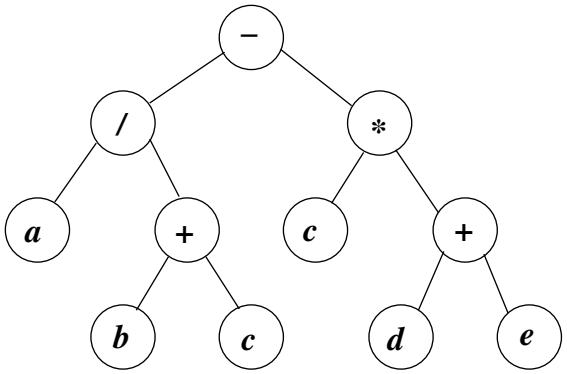
CHARACTERISTICS OF THE ALGORITHM

- 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, i.e. if the expression $a * b$ has to be evaluated using an instruction which computes $r_1 * r_2$, where r_1 and r_2 are registers, then a and b have to be necessarily loaded in r_1 and r_2 respectively.
- Generates optimal code – i.e. code with least number of instructions. It may be noted that there may be other notions of optimality. As an example the cost measure could include the cost of individual instructions.
- Complexity of the algorithm is linear in the size of the expression tree. This means that the algorithm is reasonably efficient.

SETHI-ULLMAN ALGORITHM

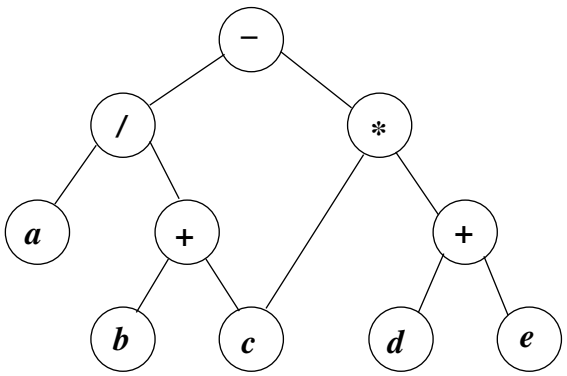
EXPRESSIONS CONSIDERED

Expressions mainly occur on the right hand side of assignment statements. We can represent an expression in the form of a tree as shown below.



This tree corresponds to the expression $a/(b+c) - c * (d+e)$.

NOTE: In the above expression tree we have not identified common sub-expressions. Otherwise we would consider the following structure.



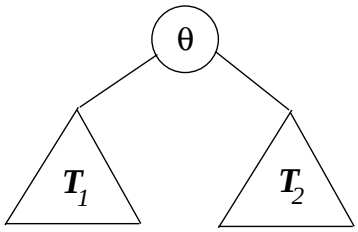
This structure is called an expression DAG (directed acyclic graph).

SETHI-ULLMAN ALGORITHM

EXPRESSION TREES

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

1. **A single vertex labeled by a name from Σ is an expression tree.**
2. **If T_1 and T_2 are expression trees and θ is a operator in Θ, then**



is an expression tree.

In the previous example
Σ = {a, b, c, d, e, ...}, and Θ = {+, −, ∗, /, ...}

TARGET MACHINE MODEL

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

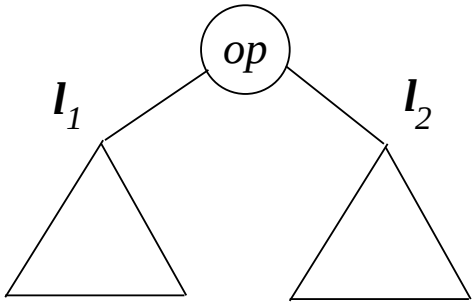
$m \leftarrow r$	**(store instruction)**
$r \leftarrow m$	**(load instruction)**
$r \leftarrow op\ m\ r$	**(the result of $r\ op\ m$ is stored in r)**
$r_2 \leftarrow op\ r_1\ r_2$	**(the result of $r_2\ op\ r_1$ is stored in r_2)**

SETHI-ULLMAN ALGORITHM

IDEA BEHIND THE ALGORITHM

The algorithm generates optimal code for a given expression tree by determining an evaluation order of the subtrees which requires *minimum number of registers.*

Given a tree whose left and right subtrees require l_1, and l_2 ($l_1 < l_2$) registers respectively as shown in the figure, what should be the order of evaluation?



Choice 1

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

The maximum register requirement in this case is

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

SETHI-ULLMAN ALGORITHM

On the other hand, we may:

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 in this case is

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

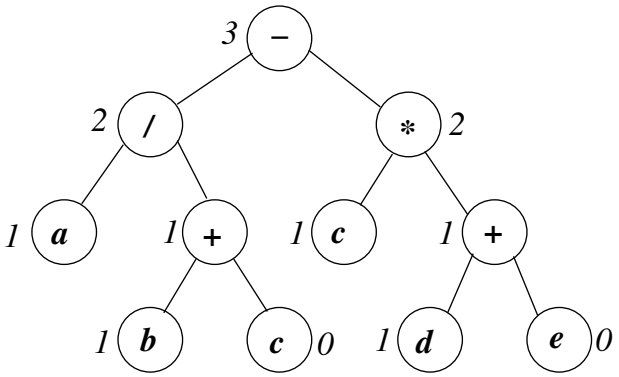
Therefore the subtree requiring more registers should be evaluated first.

SETHI-ULLMAN ALGORITHM

LABELING AN EXPRESSION TREE

Given an expression tree, we can label each node by the number of registers required to evaluate that node.

In the expression tree shown below, the label of each node has been indicated.



The left and the right leaves are labeled 1 and 0 respectively, because of the nature of our machine instructions. The left leaf must necessarily be loaded in a register before being used as an operand, whereas the right leaf can reside in memory.

LABELING ALGORITHM

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

$$label(n) = max(l_1, l_2), \text{ if } l_1 \neq l_2$$
$$= l_1 + 1, \text{ otherwise}$$

SETHI-ULLMAN ALGORITHM

ASSUMPTIONS AND NOTATIONAL CONVENTIONS

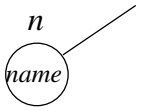
1. **The code generation algorithm is represented as a function $gencode(n)$, which produces code to evaluate the node labeled n.**
2. **Register allocation is done from a stack of register names $rstack$, initially containing $r_0, r_1, \ldots, r_k$ (with r_0 on top of the stack).**
3. **$gencode(n)$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

SETHI-ULLMAN ALGORITHM

THE ALGORITHM

We shall describe the function $gencode(n)$ by a case analysis on the type of the node n. For each case we shall indicate the appropriate fragment of the code generation algorithm below.

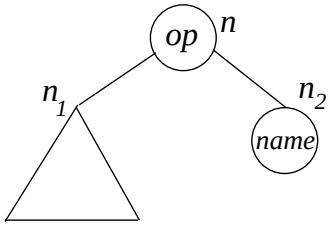
1. ***n is a left leaf:***



$$gen('\texttt{MOVE}' \;||\; name\; ',' \; top(rstack))$$

Comments: **In this case, n is named by a variable say $name$. Code is generated to load $name$ into a register.**

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

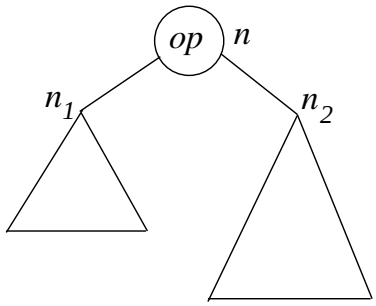


$gencode(n_1)$;
$gen(op \;||\; name \;||\; ',' \; top(rstack))$

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.**

SETHI-ULLMAN ALGORITHM

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



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

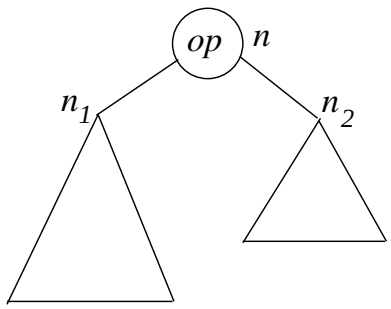
Comments: **Evaluate the right (costlier) sub-tree into the register just below the top of stack. This evaluation may possibly require some stores). Then evaluate the left sub-tree into the register on the top of stack. Now,**

- **register on top of stack contains left operand, and**
- **register below top of stack contains right operand.**

Finally, evaluate n **in the register on top of stack.**

SETHI-ULLMAN 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***

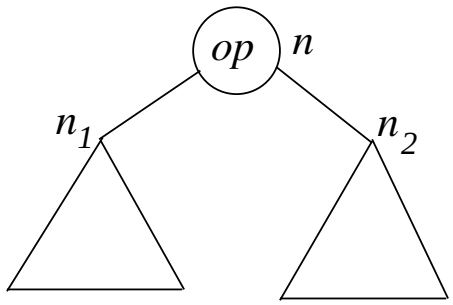


$gencode(n_1)$;
$R := pop(rstack)$;
$gencode(n_2)$;
$gen(op \mid\mid top(rstack) \mid\mid ',' \mid\mid R$;
$push(rstack, R)$

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

SETHI-ULLMAN ALGORITHM

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



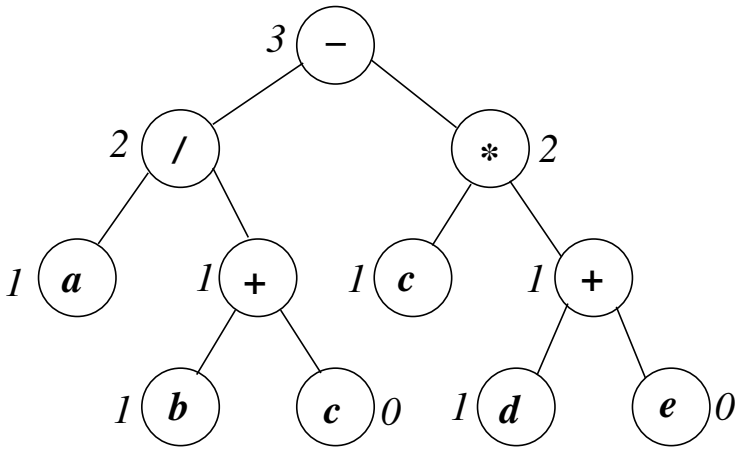
$gencode(n_2)$;
$T := pop(tstack)$;
$gen(\texttt{MOVE} \mid\mid top(rstack) \mid\mid ',' \mid\mid T)$;
$gencode(n_1)$;
$push(tstack, T)$;
$gen(op \mid\mid T \mid\mid ',' \mid\mid top(rstack)$;

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.**

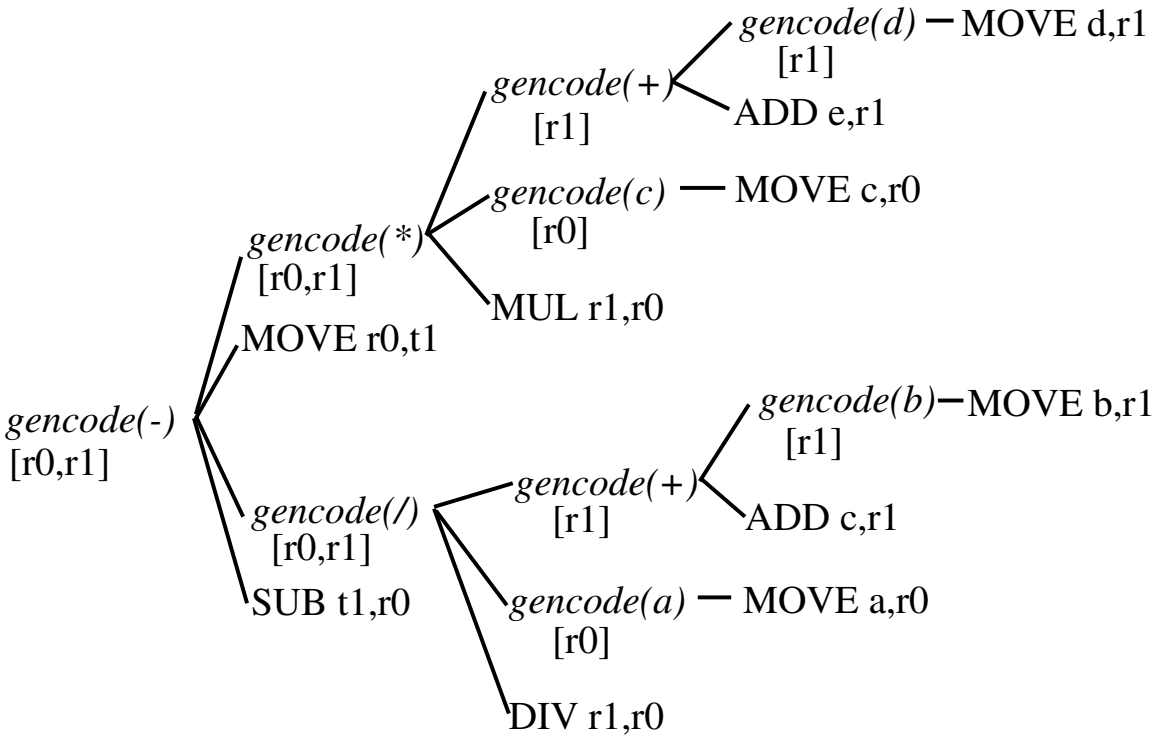
SETHI-ULLMAN ALGORITHM

AN EXAMPLE

For the example:



assuming two available registers r_0 and r_1, the calls to gencode and the generated code are shown below:



SETHI-ULLMAN ALGORITHM

PROOF OF OPTIMALITY

The algorithm is optimal because

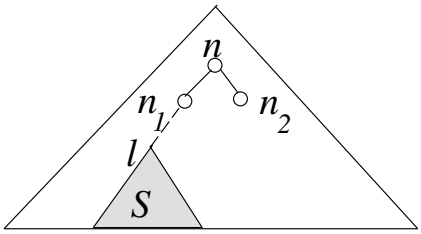
1. The number of load instructions generated is optimal.
2. Each binary operation specified in the expression tree is performed only once.
3. The number of stores is optimal.

We shall now elaborate on each of these.

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. 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?)

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.



Now reason as follows:

a. 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).

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

c. The label of n is $\geq k$.

d. 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.***

e. Therefore the number of major nodes in the modified tree is $M - 1$.

f. If we assume as induction hypothesis that the number of stores for the modified tree is at least $M - 1$, then the number of stores for the original tree is at least M.

SETHI-ULLMAN ALGORITHM

COMPLEXITY OF THE ALGORITHM

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 $O(n)$.

AHO-JOHNSON ALGORITHM

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.

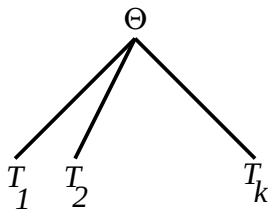
AHO-JOHNSON ALGORITHM

EXPRESSION TREES DEFINED

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

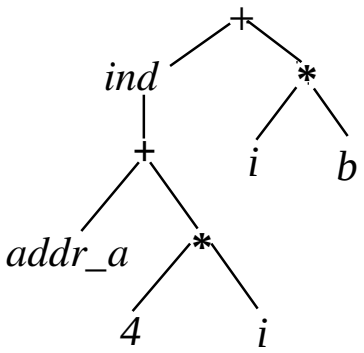
1. **A single vertex labeled by a name from Σ is an expression tree.**

1. **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.

An example of an expression tree is



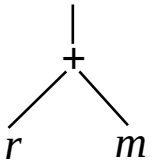
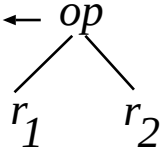
NOTATION: If T is an expression tree, and S is a subtree of T, then T/S is the the tree obtained by replacing S in T by a single leaf labeled by a distinct name from Σ.

AHO-JOHNSON ALGORITHM

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

AHO-JOHNSON ALGORITHM

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 $a[i] + i * b$

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

AHO-JOHNSON ALGORITHM

VALUE OF A PROGRAM

We need to define the value $v(P)$ computed by a program P. This is because:

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 $v(P_1) = v(P_2)$.**

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 $I_q = z \leftarrow E$, then the value of P is $v_q(z)$.

The next transparency shows an example.

AHO-JOHNSON ALGORITHM

EXAMPLE

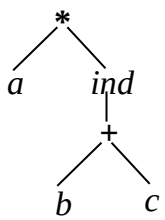
For the program shown below,

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

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

	r_1	r_2	a	b	c
before 1	U	U	a	b	c
after 1	U	U	a	b	c
after 2	+ (b, c)	U	a	b	c
after 3	+ (b, c)	a	a	b	c
after 4	+ (b, c)	* (a, ind(+ (b, c)))	a	b	c

The value of the program is



AHO-JOHNSON ALGORITHM

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 $P = I_1 I_2 \ldots I_q$ 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 $s = U_p(t)$.

AHO-JOHNSON ALGORITHM

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.

THEOREM (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 $\pi(U_P(t)) = U_Q(\pi(t))$.

AHO-JOHNSON ALGORITHM

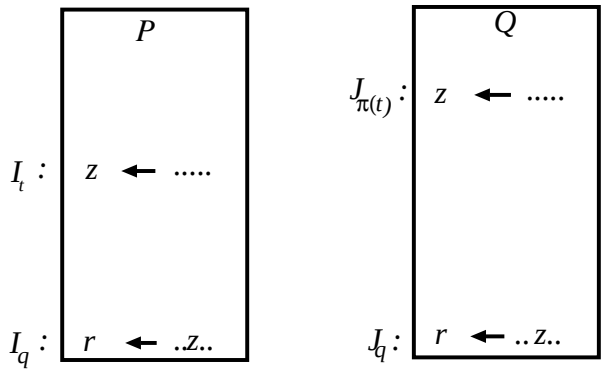
NOTES: The rearrangement theorem merely states that a rearrangement retains program equivalence, if any variable defined by an instruction in the original program is last used by the same instructions in both the original and rearranged program.

To see why the statement of the theorem is true, reason as follows:

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.**



AHO-JOHNSON ALGORITHM

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.

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

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

AHO-JOHNSON ALGORITHM

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

EXAMPLE:

$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_1 \leftarrow r_1 + r_2$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$	$r_2 \leftarrow c$
$r_3 \leftarrow r_3 + d$	$r_2 \leftarrow r_2 + d$
$r_1 \leftarrow r_1 * r_3$	$r_1 \leftarrow r_1 * r_2$

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.

This example motivates the following lemma.

AHO-JOHNSON ALGORITHM

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.**

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.**

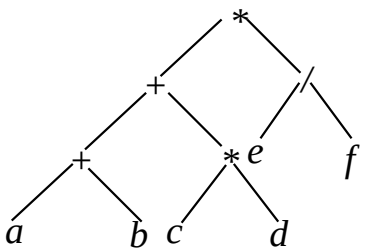
3. **Since the width of P is w, the width of the instruction, just before $r \leftarrow E$ is at most $w - 1$. Therefore a register can always be found for r in the rearranged program P'.**

AHO-JOHNSON ALGORITHM

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:



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

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.

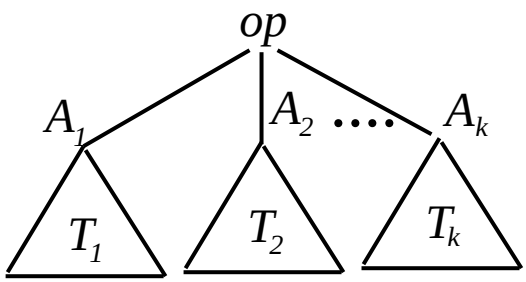
AHO-JOHNSON ALGORITHM

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 $w - i + 1$.**

Consider an evaluation of the expression tree:.



This tree can be evaluated in the order mentioned below:

AHO-JOHNSON ALGORITHM

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.

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.

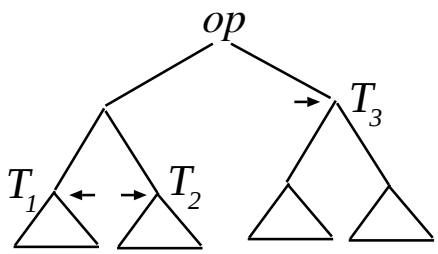
PROOF OUTLINE: Apply the technique in the previous theorem recursively to each of the P_is.

AHO-JOHNSON ALGORITHM

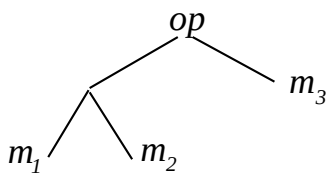
STRONG NORMAL FORM PROGRAMS

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*.

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.**



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.

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:

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

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.

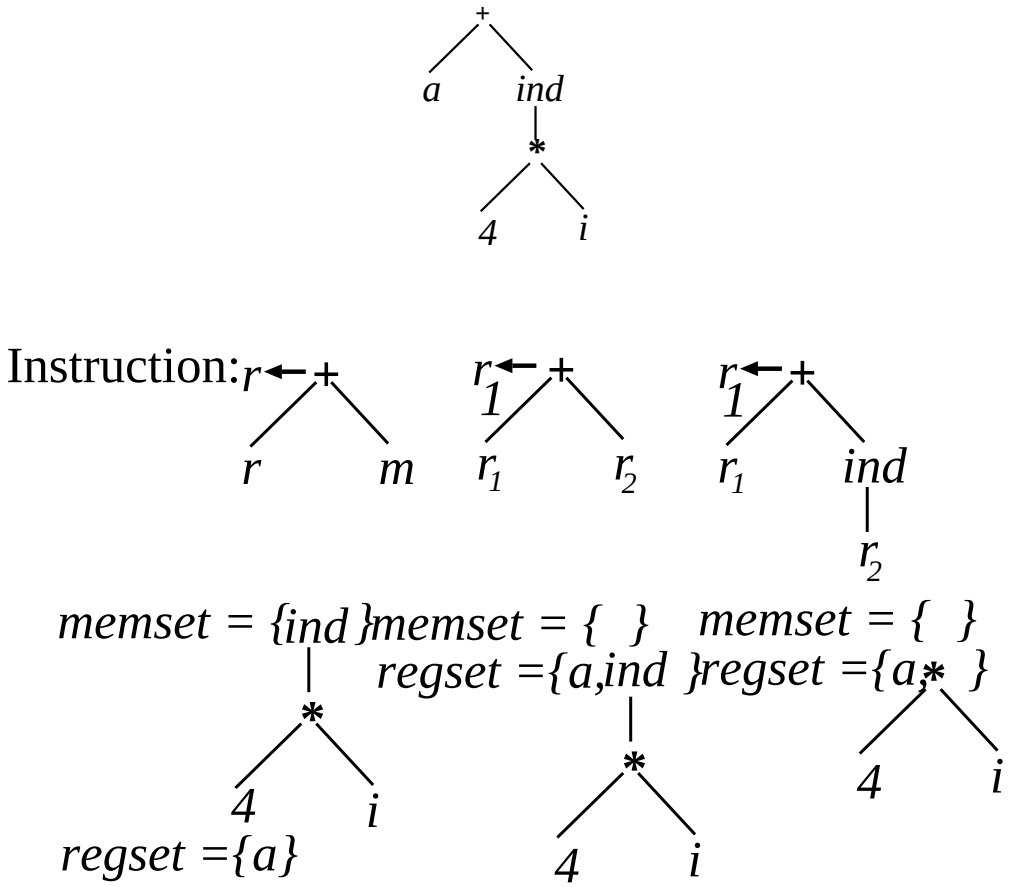
Pass 3 Actually generates code.

AHO-JOHNSON ALGORITHM

COVER

An instruction *covers a node* in an expression tree, if it can be used to evaluate the node. 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) and which should be evaluated in memory locations (memset).

EXAMPLE

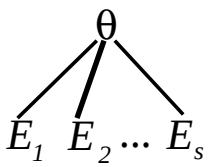


AHO-JOHNSON ALGORITHM

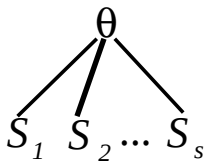
ALGORITHM FOR COVER

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 $true$.**
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

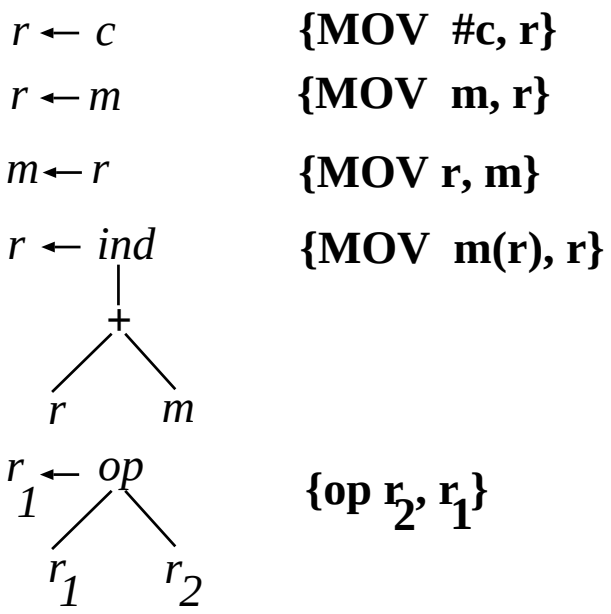
Calculates an array of costs $C_j(s)$ for every subtree S of T, whose meaning is to be interpreted as follows:

$C_j(S), j \neq 0$: is the minimum cost of evaluating S using j registers.

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

EXAMPLE

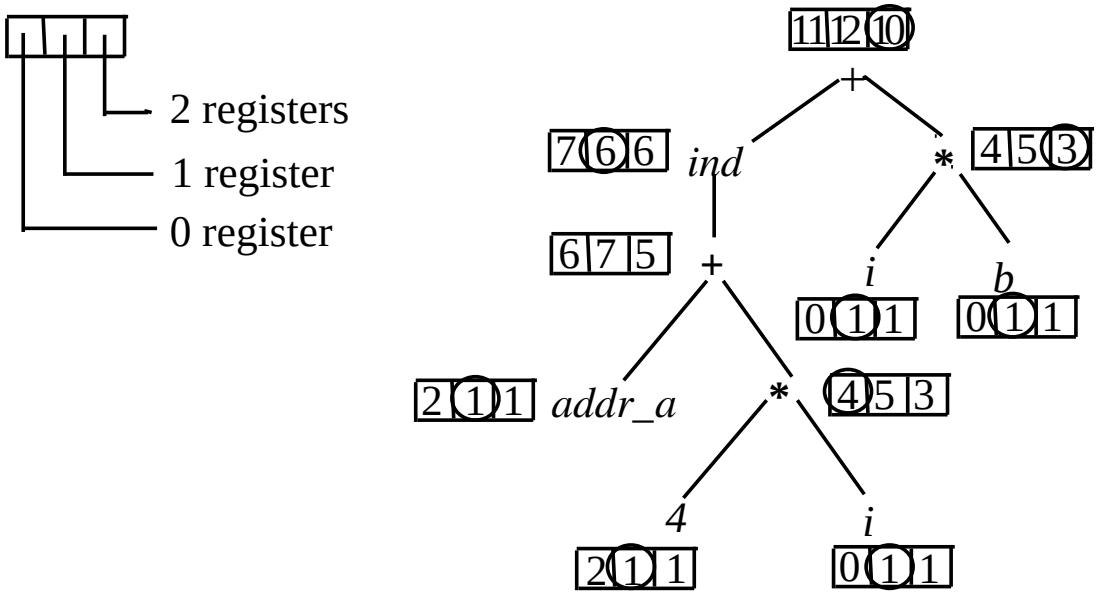
Consider a machine with the instructions shown below.



Note that there are no instructions of the form $op\ m,\ r$ or $op\ r,\ m$.

AHO-JOHNSON ALGORITHM

We show an example expression tree, along with the cost array at each node:



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,

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.**

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. 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, 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

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

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

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.**
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

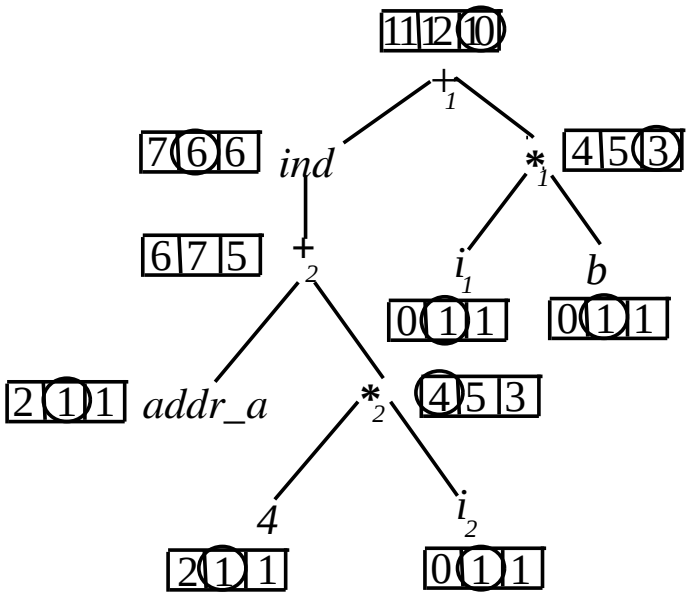
PASS 2

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



$mark(+_1, 2)$

  $mark(*_1, 2)$

    $mark(i_1, 2)$

    $mark(b_1, 1)$

  $mark(ind, 1)$

    $mark(addr_a, 1)$

    $mar\mathbf{k}(*_2, 2)$ **the covering instruction is** $m \leftarrow r$

      $mark(*_2, 2)$ **the covering instruction is** $r \leftarrow \ldots$

        $mark(4, 2)$

        $mark(i_2, 1)$

      $x_1 = *_2$ $*_2$ **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.**
- **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*}**

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.

AHO-JOHNSON ALGORITHM

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.**

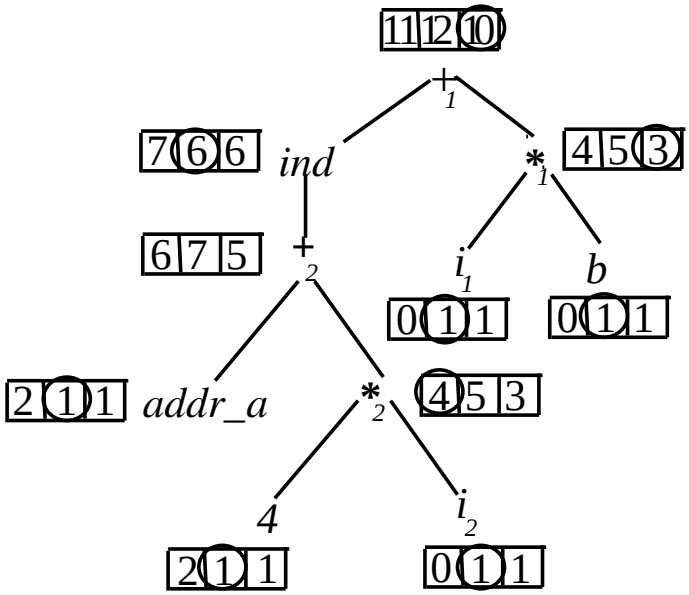
3. **If** $k = 0$, **call alloc to obtain an unused register to return.**

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.

5. **Call** $free$ **on** $\alpha_1, \ldots \alpha_k$ **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:

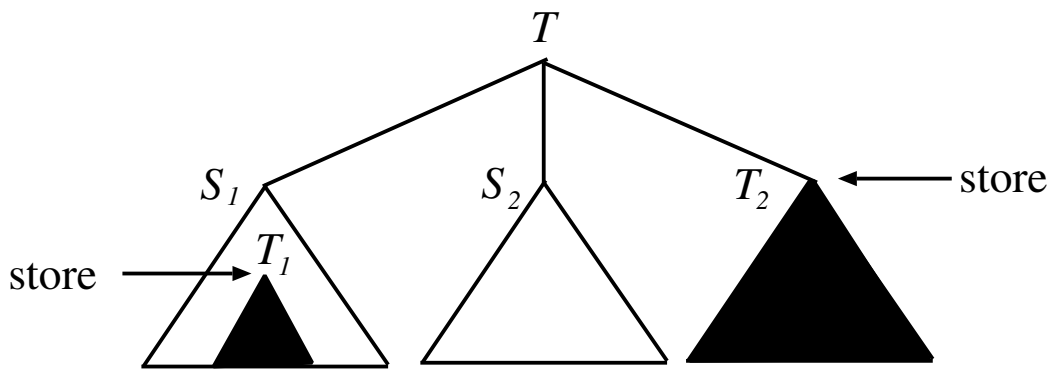


$MOVE\ \#4,\ r_1$ **(evaluate** $4 * i$ **first, since**
$MOVE\ i,\ r_2$ **this node has to be stored)**
$MUL\ r_2,\ r_1$
$MOVE\ r_1,\ m_1$
$MOVE\ i,\ r_1$ **(evaluate** $i * b$ **next, since this**
$MOVE\ b,\ r_2$ **requires 2 registers)**
$MUL\ r_2,\ r_1$
$MOVE\ \#addr_a,\ r_1$
$MOVE\ m_1(r_1),\ r_1$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

AHO-JOHNSON ALGORITHM

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 $O(n)$.

GRADED EXERCISES

1. **Augment the code generation algorithm presented in this module to incorporate the following features :**

   (a) **The parentheses in an expression**

   (b) **Non-commutative operators like '–' and '/', etc.**

2. **Show various steps in the code generation of the expression**

   $$(a + b)/(c + d)$$

   assuming two machine registers to be available.

3. **Answer question 2 above, if a single machine register were to be available.**

   Is the code generated by the algorithm for this expression 'optimal' ? If not, explain why it is not optimal.

4. **What will be the code generated by the code generation algorithm given on page 538 of the A-S-U book for the following expression :**

   $$(a + b) + (c * (a + b) + d)$$

   Explain how certain improvements in the quality of the generated code have been made by the algorithm.

5. **Assuming that the machine shown below**

   $m \leftarrow r$ **(load instruction)**
   $r \leftarrow m$ **(store instruction)**
   $r \leftarrow op\ m\ r$ **(the result of** $r\ op\ m$ **is stored in** r**)**
   $r_2 \leftarrow op\ r_1\ r_2$ **(the result of** $r_2\ op\ r_1$ **is stored in** r_2**)**

   has two general purpose registers and adequate operators, generate optimal code for the expressions shown below using Sethi-Ullman algorithm.

   a. `a / ( b + c ) - d * ( e + f )`

   b. `a + ( b + c / d * e ) / ( f * g - h * i )`

6. **Use algebraic properties (e.g. commutativity and associativity) of operators to rearrange the expressions, so that the labels at the nodes of the corresponding expression trees are minimized.**

7. **Generalise the method used in 2. Write an algorithm, which will take an expression tree and rearrange it to minimise the label at each node. Is such a rearrangement always safe?**

8. **Repeat exercise 5 using Aho-Johnson algorithm, assuming the machine model described in the transparencies.**

9. **Explain step 3 of the cost calculation procedure restated below.**

   Set $C_0(S) = min(C_0(S), C_n(S) + 1)$**, and**
   $C_j(S) = min(C_j(S), C_0(S) + 1)$**.**

   Can you construct an example to show that excluding this step might lead to non optimal code being generated.