Compiler Construction

Rationale

Language translation techniques play a vital role in many fields of Computer Technology. The need for language translators arises because the language in which one wants to express ones computational needs is significantly different from the language recognised by a real machine. Development of translators, therefore, have resulted in two important fallouts. On one hand it has a strong influence on the design of new programming languages, on the other the issue of efficient translation of programming languages is inherently linked with the design of new architectures for higher throughputs. Language translation also plays an important role in the design of command languages and user interfaces of software systems, which require knowledge of scanning and parsing techniques.

In the current context, it is essential for computer science undergraduate students to develop a thorough knowledge of the theory and techniques of language translation. This course is aimed at imparting this knowledge through a detailed treatment of the theory of language translation, and a familiarity with the use of currently available software tools for automatic generation of various modules of a language translator.

Objectives

At the end of this course on Compiler Construction, the student should be able to

1. Understand the two models of language processing, viz. compilers and interpreters, and realize their strengths and limitations.

2. Understand the influence of various features of programming languages and machine architectures on the complexity and efficiency of language translation.

3. Appreciate the role of lexical analysis in compilation, understand the theory behind design of lexical analysers and lexical analyser generators, and be able to use Lex to generate lexical analysers.

4. Be proficient in writing grammars to specify syntax, understand the theories behind different parsing strategies – their strengths and limitations, understand how generation of parsers can be automated, and be able to use Yacc to generate parsers.

5. Understand the error detection capabilities of a compiler and how to report them in a user friendly manner.
6. Understand the need for various static semantic analyses such as declaration processing, type checking, and intermediate code generation, and be able to perform such analysis in a syntax directed fashion through the use of attributed definitions.
7. Understand the issues involved in allocation of memory to data objects, and the relation of such issues with higher level language features such as data types, scope and nesting rules, and recursion.
8. Understand the key issues in the generation of efficient code for a given architecture.
9. Understand the role played by code optimisation.

Compiler Construction

Course Modules

Module 1 : Introduction to Language Processing

Module 2 : Lexical Analysis

Module 3 : Syntax Analysis

Module 4 : Error Analysis

Module 5 : Static Semantics and Intermediate Code Generation

Module 6 : Run-time Environments

Module 7 : Code Generation

Module 8 : Code Optimisation

Preface

In this module, we first introduce a simple code generator for expresssions for a simple machine architecture. This code generator has the basic features for generating appropriate instructions considering the addressibilities of the operands. The code generator also manages register usage and the saving of partial results. It then utilizes these basic features to construct a practical code generator. In the second half of this module we tackle the important issues in efficient code generation, e.g. the handling of instruction costs, register utilization and evaluation order. The Sethi-Ullman algorithm for determining the evaluation order for an expression tree is then considered. We then consider the optimal code generation for expression trees using the Aho-Johnson dynamic programming algorithm for optimal code generation.

Contents

1 Introduction

The primary concern in the compilation of expressions is the efficient utilisation of machine resources, viz. machine instructions and registers. We present the material in two parts. In the first part we present a simple code generator. This part discusses simple techniques and data structures for performing instruction selection and management of machine registers. The second part discusses algorithms for making efficient utilisation of machine resources. Note that code generation for control structures of the language has already been discussed in the module on Static Semantic analysis and Intermediate code generation(module 5).

2 A simple code generator

We make the following assumptions while describing the simple code generator :

1. All instructions in the target machine are two-operand instructions. The machine has Intel 8086-like operation codes, and has n registers named R0, R1 $\cdots$ R$n-1$. In the interest of simplicity we will generate code using a single machine register R0, however the schematic can be easily extended to the use of multiple registers.

2. All operators in an expression are commutative.

3. 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. We use a YACC-like syntax for the SDT.

We make these assumptions to simplify our discussion. However, none of these assumptions impose any fundamental constraints. Each of these assumptions can be relaxed through appropriate generalisations.

2.1 Operand descriptors

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,

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

2. 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).

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

We define *operand addressability modes* to indicate how an operand should be accessed. These have the following values :

'S'	:	Value in Storage
'R'	:	Value in Register
'AS'	:	Address in Storage
'AR'	:	Address in Register

The code to be generated for an expression depends on the addressabilities of its operands. For example, consider the statement $a := b+c$; If variables b and c are in memory, the following code would have to be generated

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

while if c is in register R0, the following code would suffice

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

We use the data structure called *Operand Descriptor* to store all details required for addressing an operand. Figure 1 shows the format of the operand descriptor. Here, the attributes could be the type and size of the operand, etc., and 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.

As already mentioned, the addressability codes could have the values 'S', 'R', 'AS' and 'AR'. Of these, the 'S' and 'R' codes are easy to imagine. The codes 'AS' and 'AR' arise in the following contexts: Addressability 'AS' (address in storage) arises for parameters of a procedure call. A list

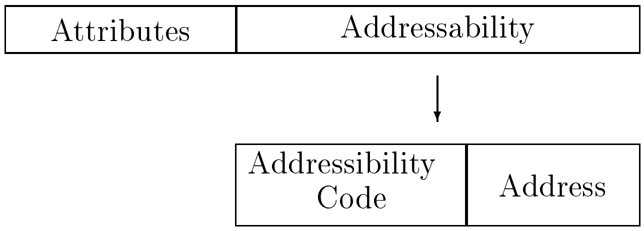


Figure 1: Operand descriptor

of addresses of the actual parameters is typically constructed in memory (possibly in an activation record). Addressability '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. For example, in the statement $x := y + info[i, j]$; the address of ***info***$[i, j]$; would be in a register after the address arithmetic.

Constructing operand descriptors

An operand descriptor is constructed when an entity is identified as a potential operand. The descriptor may be constructed in a table (viz. Symbol table entry of a variable) or elsewhere. For simplicity we will assume operand descriptors to be stored in an array called ***Operand_descr***. Thus, the sequence of steps while generating code for the expression $a * b + c \cdots$ would be as follows :

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 operator '+').

Figure2 shows a skeletal code generator using YACC-like syntax. The attribute of 'id' is the symbol table entry of the id, while the attributes of Factor, Term and Exp are entry numbers in the array ***operand_descriptor***. The routine 'build_descriptor' builds a descriptor for an id, by picking up its type and address from the symbol table entry. The routine 'codegen' generates the code for evaluating an operation. It checks the addressability of both operands by referring to their operand descriptors, and generates

```
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, addressability, address) of opd1;
      return i;}
codegen  (operator, opd1, opd2);{
      if  opd1.addressability = 'R'  then
                        /* Code generation – case 1 */
                if  operator = '+'  generate  ( 'ADD opd2, R0' );
                else  if  operator = '*' then  ···
      else  if  opd2.addressability = '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 descriptor for the value in register R0 */
      i = i + 1;
      operand_descr[i] = (< type >, 'R', R0);
      return i;}
```

Figure 2: A skeletal code generator

appropriate code. In the common part, it creates a new descriptor for the result of the operation, and returns its entry in ***operand_descriptor*** as its result. This value is noted as the attribute of the LHS nonterminal in the production.

Figures 3 and 4 show code generation for the expression $a + b + c$. In each parsing action, the attribute of the LHS nonterminal is shown as a superscript. As remarked earlier, these attributes are the entry numbers in ***Operand_descriptor***. Thus in the parsing action $\mathbf{F}^1 \leftarrow a$ depicted in Figure 3, the value represented by the nonterminal F is described by the operand descriptor in the first entry of ***Operand_descriptor***. Notice how the addressability of an operand leads to the generation of appropriate instruction. During the reduction $\mathbf{E}^3 \leftarrow \mathbf{E}^1 + \mathbf{T}^2$, the operands described by operand descriptors 1 and 2 are both in memory. Hence it is necessary to load one operand into R0 before performing the addition. During the reduction $\mathbf{E}^5 \leftarrow \mathbf{E}^3 + \mathbf{T}^4$, however, the operand described by operand descriptor 4 is already in R0. Thus, no MOV instruction is generated for evaluating the second '+' operator, a single ADD instruction suffices.

2.2 Handling partial results

Figure 5 shows a step in the code generation for the expression $a * b + c * d$ wherein code has been generated for the subexpression $a*b$. The result of this evaluation is in register R0. The string $c * d$ has been parsed, and descriptors have been built for c and d. The code generator has now paused prior to the reduction T $\leftarrow$ T $*$ F. The code about to be generated is thus

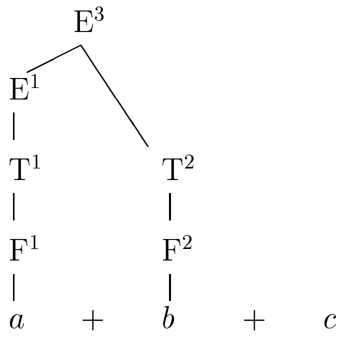
```
MOV     C, R0
MUL     D, R0
```

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. This leads to an obvious question: When should the contents be saved in a memory location? A naive approach would be to save the contents of the register after every operation. This is obviously a very expensive solution as can be seen from the following code for the expression $a + b + c$:

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

Parsing actions		*Operand_Descr* ($i = 0$ initially)		
$F^1 \leftarrow a$	1	int	S	Ad(a)
$T^1 \leftarrow F^1$		– Same –		
$E^1 \leftarrow T^1$		– Same –		
	1	int	S	Ad(a)
$F^2 \leftarrow b$	2	int	S	Ad(b)
$T^2 \leftarrow F^2$		– Same –		
	1	int	S	Ad(a)
	2	int	S	Ad(b)
$E^3 \leftarrow E^1 + T^2$	3	int	R	R0

Parse tree :



Generated code :

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

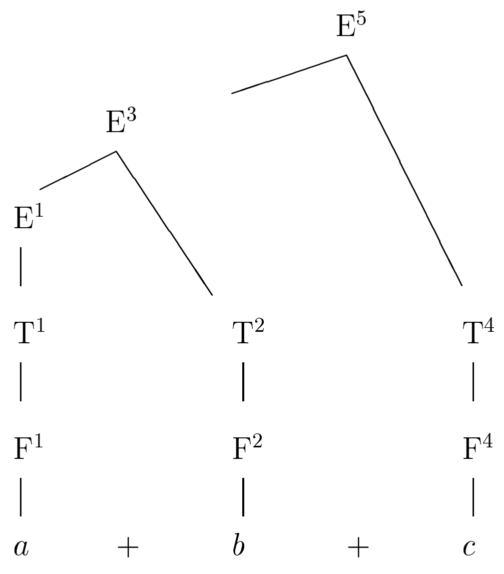
Figure 3: Code generation for $a + b + c$

Parsing actions		*Operand_Descr*		
	1	int	S	Ad(a)
	2	int	S	Ad(b)
	3	int	R	R0
$F^4 \leftarrow c$	4	int	S	Ad(c)

$T^4 \leftarrow F^4$ – Same –

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

Parse tree :

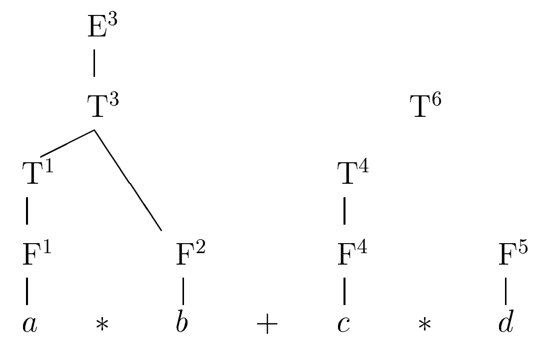


Generated code : ADD C, R0

Figure 4: Code generation for $a + b + c$ (contd.)

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

Parse tree :



Generated code :

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

Figure 5: Partial results during code generation

MOV	R0, Temp_1
MOV	Temp_1, R0
ADD	C, R0

Here, contents of R0 were saved and reloaded for no good reason! The best approach would be to ensure that a register is freed only when all registers are occupied by partial results and a register is needed to evaluate an independent subexpression, i.e. when we are about to generate code for a subexpression which does not use the contents of any of the registers. We present a solution for this problem in the next section.

2.3 Register descriptors

A register descriptor describes the contents of each register. The fields in the descriptor are:

Occupied ?	Descriptor No.

The ***Occupied ?*** flag indicates whether the register currently contains a useful result. It has the values ***Occupied*** or ***Free***. If the register is occupied, the ***Descriptor number*** field points to the operand descriptor of the value currently contained in the register. Register descriptors are stored in an array called ***Register_Descr***, which contains as many elements as the number of registers in the machine. (In our case, there would be only one register descriptor. However, having an array makes it possible to generalise on this scheme easily.)

Figure 6 shows the contents of the operand and register descriptors during the compilation of the expression $a*b+c*d$. The result of the addition is described by the operand descriptor in the third position of ***operand_descriptor***. Therefore, the register descriptor contains the flag ***Occupied***. The second field of the register descriptor contains '3' to indicate that the register contains the value described by this descriptor.

We can now return to the question of when should a partial result be transferred from a register to a memory location. We can check the operands of every operation to see if any of them reside in a register. If so, we know that the operation can be performed straightaway between the contents of this register and the other operand. If none of the operands are in the register, then we need a free register to perform the operation. We can

Operand_Descr

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

Register_Descr

R0	Occupied	3

Figure 6: Register descriptor usage

check the status flags of all registers to see if any of them is free. If not, we must free a register by storing its contents in the memory.

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. To save a partial result we must do two things:

1. Generate an instruction

   MOV   *reg_no*, TEMP_j

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

2. Change the descriptor in

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

   to

$<$ *type* $>$	S	Ad(TEMP_j)

   and set *Register_descr*[*reg_no*] = (Not occupied, —).

Figure 7 contains the complete code generation routine. The part of code marked with '$\Leftarrow$' now contains the provision for saving partial results in the temporary locations. Figure 8 shows the steps in code generation for the expression $a * b + c * d$. Note how in step 2 we have the situation that the register is occupied, and the operands of the operation for which we are about to generate code are both in memory (see the operand descriptors 4 and 5). A MOV instruction is now generated to put the contents of the register in TEMP_1. The addressability in operand descriptor 3 is changed to indicate that the operand is no longer in the register.

```
codegen  (operator, opd1, opd2);
{
    if  opd1.addressability = '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 */
    ...
}
```

Figure 7: Complete code generation routine

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
4	int	S	Ad(c)
5	int	S	Ad(d)

Occupied	3

Generated code : MOV A, R0
MUL B, R0

Step 2 : Situation after generating a STORE instruction :

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)

Not Occu.	—

Generated code : MOV R0, $TEMP_1$

Step 3 : Situation after code generation for '+' :

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)
6	int	R	R0
7	int	R	R0

Occupied	7

Generated code : MOV C, R0
MUL D, R0
ADD $TEMP_1$, R0

Figure 8: Partial result handling in the expression $a * b + c * d$

2.4 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 $\ldots$ TEMP_j $\ldots$ A natural question is whether it is essential to use a distinct temporary location for each partial result. In other words, can we minimise the number of temporary locations required ?

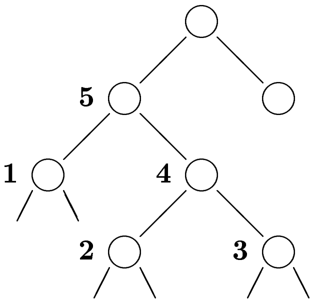
A temporary location TEMP_j used say, in statement S_i, can surely be re-used in statement S_{i+1} since all statements are independent of one another. An examination of the code generation approach also reveals that TEMP_j can be re-used within statement S_i itself in a *last-in-first-out* (LIFO) manner in the unoptimised code for an expression. This re-use is possible because of two reasons:

1. **Since we do not identify common sub-expressions, 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.

2. **Expression trees are a natural (LIFO) structure. For example, consider the following expression tree**

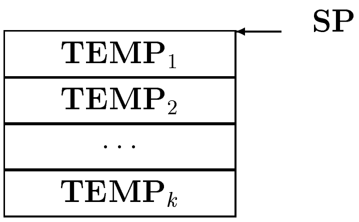
   When evaluated left-to-right, value of node 1 would be saved *before* that of node 2 (in order to evaluate node 3), and would be used *after* that of node 2.

How shall we organize the temporary locations to utilize the re-use characteristics mentioned above ? Let us assume that we have some k temporary locations. Now,

1. The temporary names $\mathrm{TEMP}_1, \ldots \mathrm{TEMP}_k$ can be put into a stack called *tempstack* at the start of compilation of a statement.

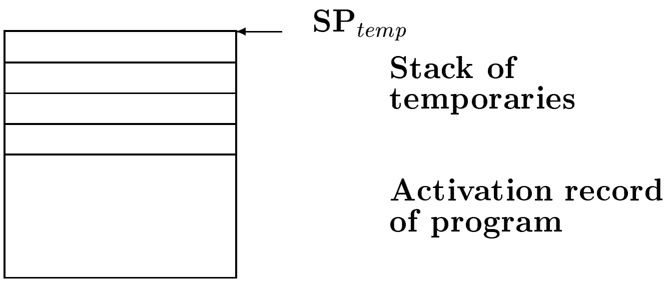
2. Whenever the compiler needs to save a partial result into the memory, it pops a temporary location name off *tempstack*.

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*.



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

The drawbacks of this scheme are that we need to know the value of k, and we need to use distinct names TEMP_1, ... TEMP_k. An alternative which overcomes both these problems is as follows: Use an anonymous stack of temporaries ***on top*** of the activation record of a program, viz.



Now, there is no need for distinct names for the temporaries. 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.

In un-optimised code for expressions not involving procedure calls, we can ensure the stack behaviour of the temporaries by saving, whenever necessary, the partial result which is the ***earliest computed result*** still in a register. In the simple code generator developed so far, this can be easily achieved using the array of register descriptors. In optimised code, a partial result may be used many times (e.g. a common sub-expression). Hence,

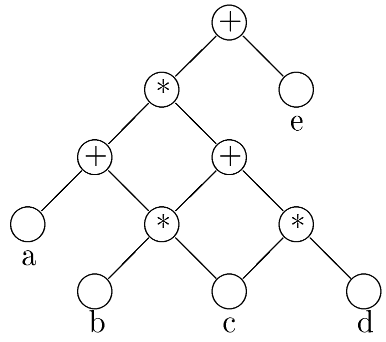


Figure 9: An expression DAG

1. **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.**

2. **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.)**

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

2.5 Practical code generation

In practice, expressions are not always trees, they can be DAG's, i.e. operations can have common nodes. In an expression DAG, a common leaf is a variable/constant used more than once whereas a common internal node represents a shared sub-expression. For example, in Figure 9, c is a shared operand, while $b*c$ is a common sub-expression within this expression. Handling of such multiple uses of an operand requires a change in our basic approach to the handling of operands during code generation. For example, in Figure 9 c may be loaded into a register while evaluating the expression $b*c$. When it is referenced again as an operand in $c*d$, it can be used from the register instead of the memory location. Thus, the operand descriptor for c constructed while generating code for $b*c$ should be reused while generating code for $c*d$. To enable this, we abandon the idea of generating a new descriptor for a variable when it is to be used as an operand. Instead,

we maintain a single descriptor for it ***in its symbol table entry.*** Whenever it is to be used as an operand, we refer to this descriptor to determine its addressability. The entry for a variable v now looks like this before start of code generation :

#25	Ad(v)		$< type >$	S	Ad(v)

↑ Descriptor

The descriptor is changed to

#25	Ad(v)		$< type >$	R	R0

↑ Descriptor

when v is loaded into a register, or when code is generated for the assignment

$$v \ := < exp >;$$

where $< exp >$ is evaluated in register R0.

The outline of our algorithm now looks as follows:

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. This ensures that a single descriptor exists for each 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. This will help us to determine whether we need to maintain this value any longer. When the last use is processed, the partial result can be destroyed.
2. Delay the Store instructions for assignments, i.e. do not generate a Store instruction immediately on processing an assignment. Do this later when the register is required for some other purposes.

As an example of step 2, consider the assignments

$$a \ := \ b \ * \ c \ ;$$
$$x \ := \ a \ ;$$

The values of a and x are held in registers at the end of processing the assignments. In fact, we can extend the idea to hold their values in the same register! (This is achieved while processing the second assignment by simply noting the fact that x and a have the same value.) This has two consequences :

1. Future references to variables a and x can be serviced out of the register. This reduces Load / Store instructions across statements.

2. Now, while freeing a register, save its contents into a temporary location (if it contains a partial result) and / or into the memory locations allocated to all variables whose value it contains (in case of assignments).

In above example, two registers contain values of expression $b * c$, and variables a, x. If the register containing the value of $b * c$ is to be freed, then its value can be stored in a temporary location. If the register containing the values of a and x is to be freed, its contents should be stored into the memory locations allocated to a and x. To achieve the latter, we extend the register descriptor to contain a list of descriptors instead of a single descriptor, viz.

Occupied ?	List of Descriptor Nos.

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

Now, to integrate these aspects into the code generation routine, we define a function called ***getreg*** which determines ***where*** to evaluate an expression, viz.

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

This function makes this decision and returns this information in L. The algorithm for function ***getreg*** is now as follows:

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.

3 Sethi-Ullman Algorithm

We shall start by pointing out the salient features of the algorithm. The algorithm generates code for expression trees and not dags. This means that common subexpressions are not identified. The target machine model is simple. It has

- a load instruction,
- a store instruction, and
- binary operations involving either a register and a memory, or two registers.

The algorithm also 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. The algorithm 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. Finally, the complexity of the algorithm is linear in the size of the expression tree. This means that the algorithm is reasonably efficient.

Expressions mainly occur on the right hand side of assignment statements. We can represent an expression in the form of a tree. As an example, the expression $a/(b+c) - c*(d+e)$ can be represented by the tree shown in figure 10.

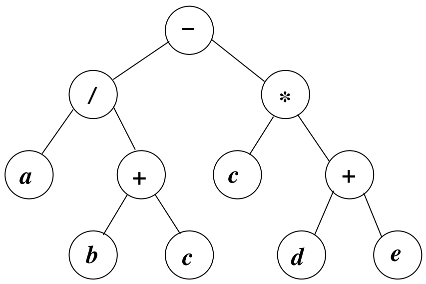


Figure 10: An expression tree

Note that in the above tree we have not identified common sub-expressions. Otherwise we would consider the structure shown in figure 11. The implication is that if our instruction required both operands to be loaded in registers, the fact that it is loaded once for evaluating the $+$ operator would be possibly used to avoid the load for the $*$.

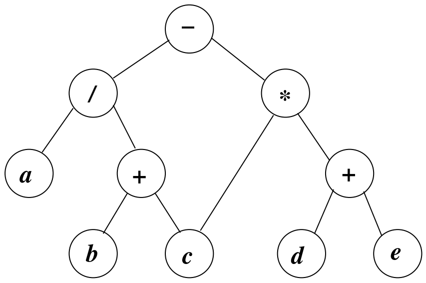


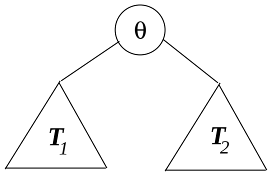
Figure 11: An expression DAG

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

Formally, the expressions that we intend to deal with can be captured as follows. Let Σ be a 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 $\Sigma = \{a, b, c, d, e, \ldots\}$, and $\Theta = \{+, -, *, /, \ldots\}$

As far as the target machine is concerned, we assume a machine with finite set of registers $r_0, r_1, \ldots, r_k$, and a set of memory locations. The instructions are 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)

3.1 Basic idea

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 figure 12, what should be the order of evaluation of the two subtrees?

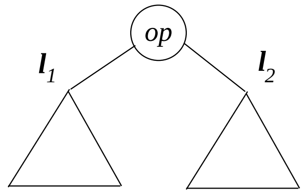


Figure 12: An expression tree with two subtrees

There are two possible choices. One can either (i) evaluate the left subtree first, leaving the result in a register. During this evaluation we shall require up to l_1 registers. And then (ii) evaluate the right subtree. During this we might require up to $l_2 + 1$ registers (l_2 registers for evaluating the right subtree and one register to hold the value of the left subtree). Clearly, the maximum register requirement in this case is $max(l_1, l_2 + 1) = l_2 + 1$

On the other hand, we may (i) evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require up to l_2 registers. And (ii) evaluate the left subtree. During this, we might require up to $l_1 + 1$ 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.*

Given an expression tree, we can label each node by the number of registers required to evaluate that node. In the expression tree shown in figure 13, the label of each node has been indicated.

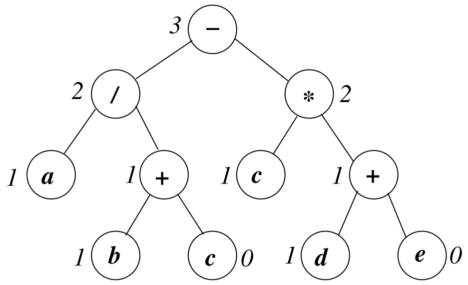


Figure 13: Labeling of expression tree

In this labeling scheme, 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. The labeling scheme visits the tree in post-order. For every node visited, it:

1. Labels each left leaf by 1 and each right leaf by 0.

2. If the labels of the children of a node n are l_1 and l_2 respectively, then

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

3.2 The Algorithm

Before describing the algorithm, we shall first explain the 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.

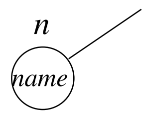
6. gen is a function with variable number of arguments. It evaluates its non-quoted arguments, concatenates the arguments in order, and returns the resulting string.

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.

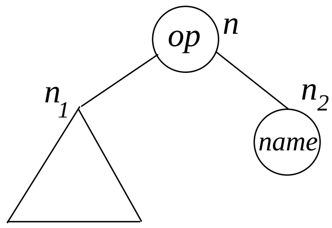
1. ***n is a left leaf:***

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

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



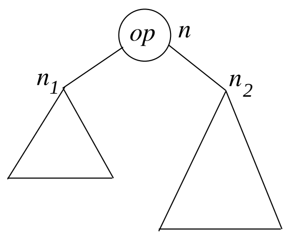
2. ***n's right child is a leaf:***



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

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.**

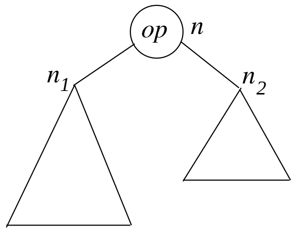
3. ***The left child of*** n ***requires lesser number of registers. This requirement is strictly less than the available number of registers***



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

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, the register on top of stack contains left operand, and the register below top of stack contains right operand. Finally, evaluate** n **in the register on top of stack.**

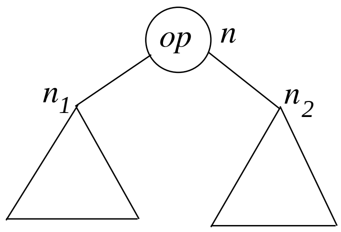
4. ***The right child of n requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers***



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

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

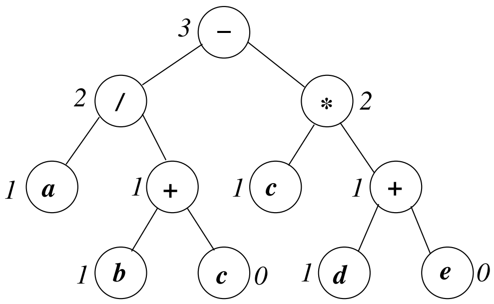
5. ***Both the children of n require registers greater or equal to the available number of registers.***



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

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 a nd n into the register on the top of the stack.**

We shall illustrate the working of the algorithm with an example. For the example:



assuming two available registers r_0 and r_1, the calls to gencode and the generated code are shown in figure 14.

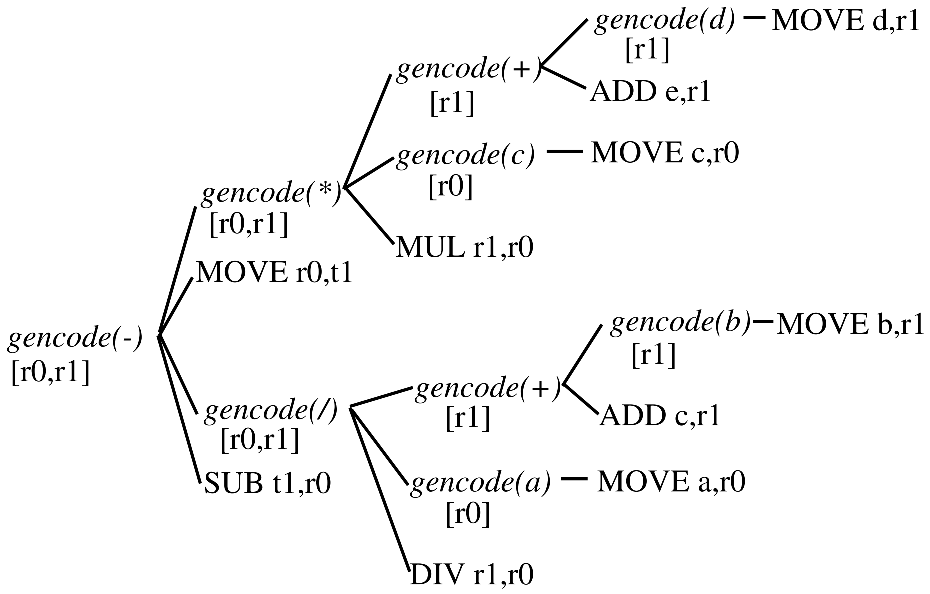


Figure 14: Calls to *gencode* and code generated for the example

3.3 Analysis of Sethi-Ullman algorithm

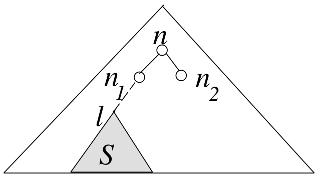
We first show that this algorithm produces optimal code, i.e. code with least number of instructions. This is so because (i) the number of load instructions generated is optimal, (ii) each binary operation specified in the expression tree is performed only once, and (iii) the number of stores is optimal. 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 we can reason as follows. 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). Now the (modified) label of n_1 might have decreased after replacing the subtree by the leaf, but the label of n_2 remains unaffected ($\geq k$, the available number of registers). Therefore the label of n is still $\geq k$. It follows that though 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*, and the number of major nodes in the modified tree is $M-1$. Now, 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.

What is the complexity of Sethi-Ullman 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 $O(n)$.

4 Aho-Johnson algorithm

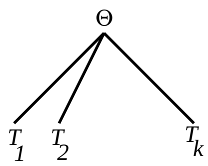
Like Sethi-Ullman algorithm, this algorithm also considers expression trees instead of DAGs. However, the algorithm differs from the earlier one in its ability to handle a wider class of target machine. The target machine model is general enough to represent a large class of machines. Represented as 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

As in the case of Sethi-Ullman, the algorithm does not use algebraic properties of operators. This algorithm is also optimal, where, once again, the cost measure is the number of instructions in the code. And finally, the complexity of the algorithm is linear in the size of the expression tree.

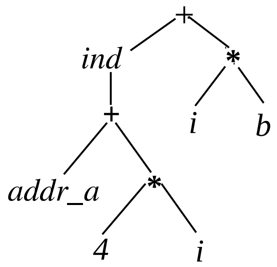
The expression trees considered are more general than those considered by Sethi-Ullman algorithm. Formally, let Σ be a countable set of operands, and Θ be a finite set of operators. Then,

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.



4.1 The machine model

The machine has n general purpose registers and no special registers. It has a set of memory locations. Instructions are of the form:

a. $r \leftarrow E$, r is a register and E is any expression tree whose operators are from Θ and operands are registers, memory locations or constants. This is what gives the target machine its generality. Further, r ***should be one the register names occurring (if any) in*** E.

b. $m \leftarrow r$, a store instruction.

Figure 15 shows an example of a machine. The instructions of the machine have also been represented in a tree form.

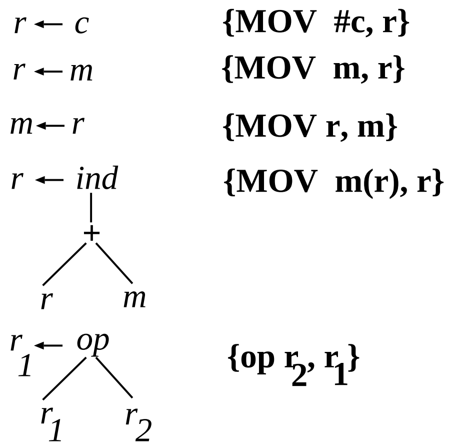


Figure 15: Example of a machine

4.2 Machine program

A ***machine program*** consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$. As an example, the machine program below evaluates $a[i] + i * b$.

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

4.2.1 Value computed by a program

We need to define the value $v(P)$ computed by a program P. This is because 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. Further, 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, which is defined in the following way. We 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 $v_t(z) = v_{t-1}(z)$.

Now we can define the value of a program as the value computed in the last instruction. In other words, If $I_q = z \leftarrow E$, then the value of P is $v_q(z)$.

We illustrate this with an 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 as shown in figure 16. U signifies that the value is undefined. The value of the program is shown in figure 17.

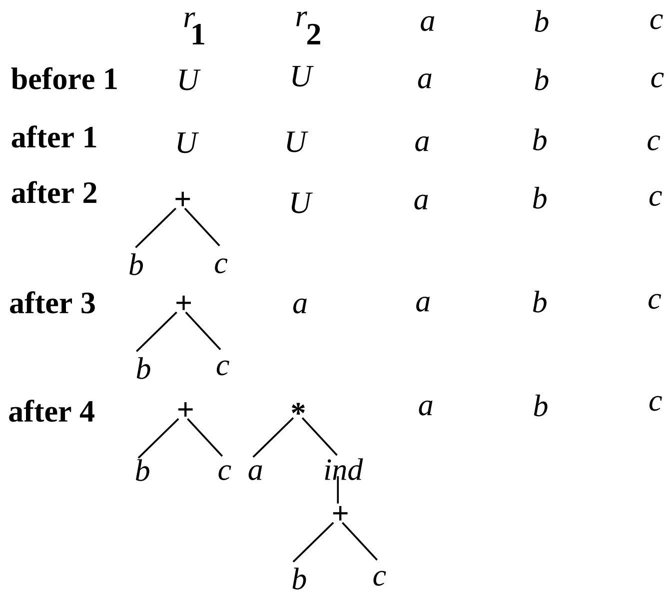


Figure 16: Value of registers and memory locations after each instruction of the example program

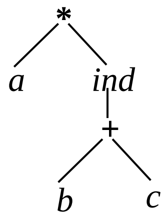


Figure 17: Value of the program

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. Since we aim to generate optimal programs, it is reasonable to assume that our programs do not have any useless instructions.

4.2.2 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 $s = U_p(t)$. However notice that if we restrict our attention to expression trees instead of DAGs, then a register or memory location defined in an instructions is used just once.

4.3 Rearrangability of programs

We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in a certain form called ***strong normal form.*** We shall define the notion of strong normal form shortly. Why is this result important? This is because while trying to search for an optimal program, our algorithm will consider only those programs which are in strong normal form. The above result assures us that by doing so, the algorithm will 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 $\pi(U_P(t)) = U_Q(\pi(t))$.

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 using figure 18 as reference.

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.

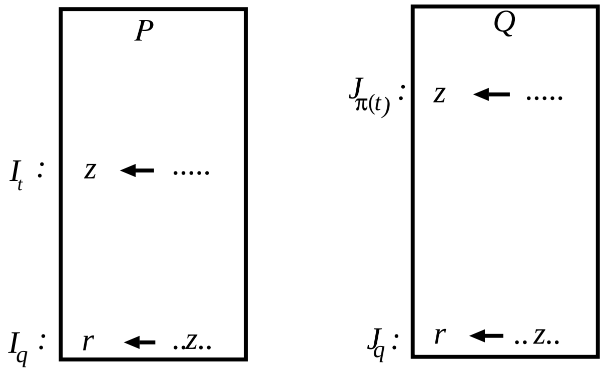


Figure 18: A program and its rearrangement

4.3.1 Width

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

Roughly an instruction has width w, if w registers are live when the instruction is executed, i.e. these registers contain values which will be used at a later instruction. For instance, the instruction I_t shown below has a width 2, since the values in register r_1 and r_2 are used after I_t.

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

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

Now a program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers. As an example, consider the program shown below:

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

$r_1 \leftarrow r_1 * r_3$ $\qquad\qquad$ $r_1 \leftarrow r_1 * r_2$

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.

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 can construct an equivalent program P', with the same length as P, which uses only registers in R.

Why does the lemma hold? Firstly, can assume that the renaming scheme algorithm is consistent, that is, when a variable which is defined is relabeled, its use is also relabeled. Now suppose 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 right hand side 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.

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'.

4.3.2 Contiguity and strong contiguity

Can one decrease the width of a program? For ***storeless programs***, there is an arrangement which has minimum width. As an example, all the three programs P_1, P_2, and P_3 compute the expression tree shown in figure 19.

$\underline{P_1}$	$\underline{P_2}$	$\underline{P_3}$
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$

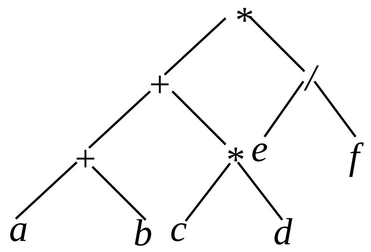


Figure 19: An example expression tree

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

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. The next theorem shows that a program can be rearranged into an eqivalent program in contiguous form whose width is no greater than the original program.

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 $w - i + 1$.**

To see this consider an evaluation of the expression tree shown in figure 20. This tree can be evaluated in the order mentioned below:

1. **Q computes the entire subtree T_1 first using P_1. In the process all the w registers could be used.**

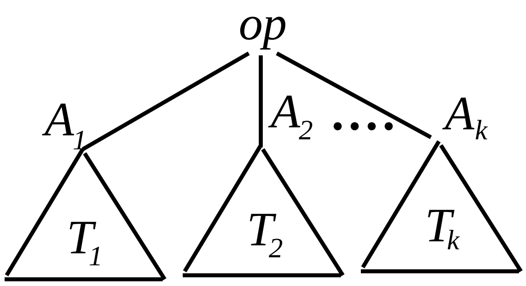


Figure 20: Skeleton of an expression tree

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 $P_1 \ldots P_k J_q$ which satisfies the above theorem 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.

This is easy to see. Just apply the technique illustrated in the proof of the previous theorem recursively to each of the P_is.

4.3.3 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 the tree shown in figure 21, 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.**

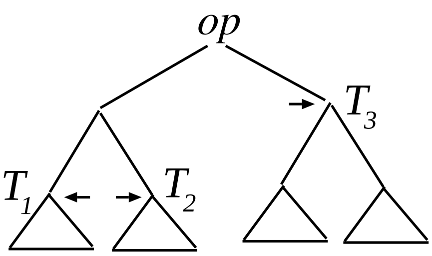


Figure 21: An expression tree which requires stores at the nodes marked with an arrow.

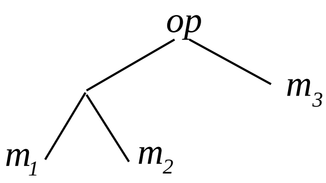


Figure 22: The same expression tree with the subtrees T_1, T_2 and T_3 replaced by leaves.

4. **Compute the resulting tree shown in figure 22 using a storeless program P_4.**

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

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.

Here is a proof outline of the lemma. Let I_f be the first store instruction of P. We can Identify the instructions between I_1 and I_{f-1} which do not contribute towards the computation of the value of I_f and shift these instructions, in order, after I_f. 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. Now we can repeat this process for the program Q.

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 mo st w, and

3. Q is in strong normal form.

Given a program, we can first apply the previous lemma to get a program in normal form and convert each P_i to strongly contiguous form. Observe that none of the above transformations increase the width or length of the program. This proves the theorem.

4.4 The algorithm

The algorithm makes three passes over the expression tree. In pass 1 it 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. In pass 2 it identifies the subtrees which must be evaluated in memory locations. Finally, in pass 3 it actually generates code.

4.4.1 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). As an example, the expression tree shown in figure 23 can be covered by each of the instructions in the same figure. The corresponding regsets and memsets are also shown.

We now present the algorithm which decides whether an instruction covers a given expression tree.

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 $true$.

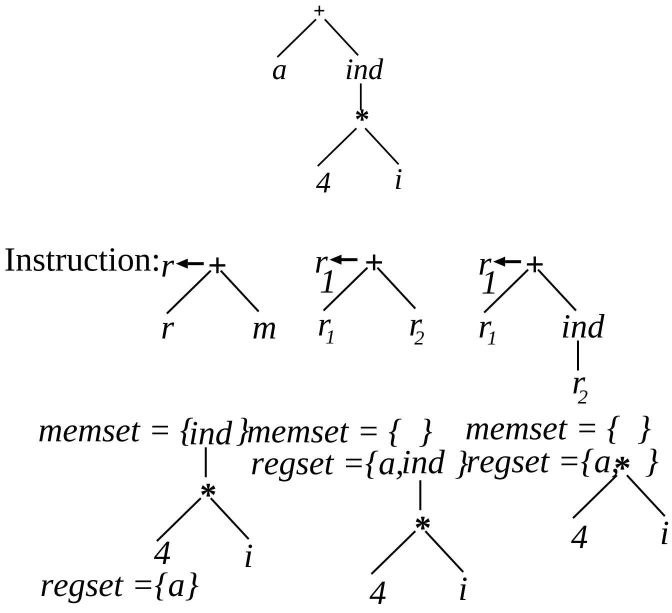
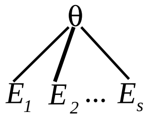


Figure 23: An expression tree and some of its covering instructions



3. If E has the form

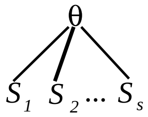
then, if the root of S is not θ, return $false$. Else, write S as

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

4.4.2 Pass 1

We now discuss the main activity of pass 1, which is to calculate 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.**



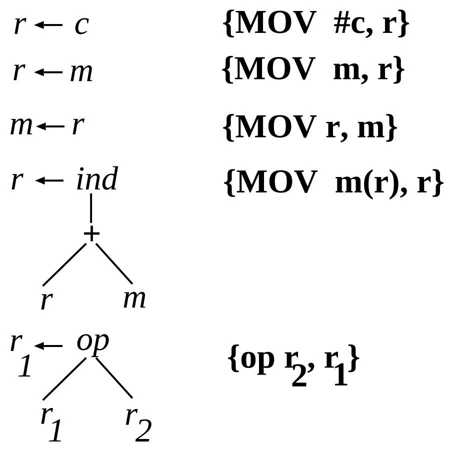


Figure 24: An example machine

Consider a machine with the instructions shown in figure 24. Note that there are no instructions of the form *op* $m,\ r$ or *op* $r,\ m$. We show an example expression tree, along with the cost array at each node in figure 25. 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. We explain in details how to calculate the cost at each node. 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.

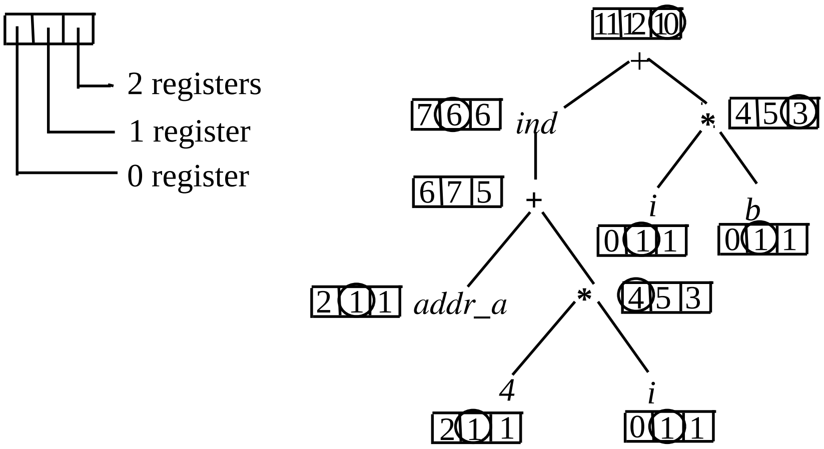


Figure 25: Calculating array of costs

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 $C[0] + 1 = 5$.

We now present pass 1 of the algorithm.

Pass 1

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
$C_j(S) = min(C_j(S), C_0(S) + 1)$.

We explain the various steps in the algorithm. 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, and 1 is the cost of the instruction at the root. In step 3, $C_0(S) = min(C_0(S), C_n(S) + 1)$ is the cost of evaluating a node in memory location by first using n registers and then storing it, and $C_j(S) = min(C_j(S), C_0(S)+1)$ is the cost of evaluating a node by first evaluating it in a memory location and then loading it.

The algorithm also records at each node, the minimum cost, the instruction which resulted in the minimum cost, and the permutation which resulted in the minimum cost.

4.4.3 Pass 2

Pass 2 of the algorithm marks the nodes which have to be evaluated into memory. The function $mark$ shown below 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.

Pass 2

Calls function $mark(T, n)$, where $mark$ is defined as

$mark(S, j)$;

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.

When the function $mark$ is run on the expression tree shown earlier, the sequence in which nodes are marked is shown below. The two $*$ nodes have been distinguished by subscripting them.

$mark(+_1, 2)$
    $mark(*_1, 2)$

$mark(i_1, 2)$
$mark(b_1, 1)$
$mark(ind, 1)$
$mark(addr_a, 1)$
$mar\mathbf{k}(*_2, 2)$ 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

4.4.4 Pass 3

In this pass the algorithm generates code for the subtrees rooted at $x_1, \ldots x_s$ marked by pass 2, 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 (i) $alloc$, which allocates a register each time it is invoked from a pool of registers, and (ii) $free$, which frees a register and returns it to the pool.
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 next.

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 $code(S, j)$.**

Once again, we show how this pass works by generating code for the earlier example which we reproduce in figure 26

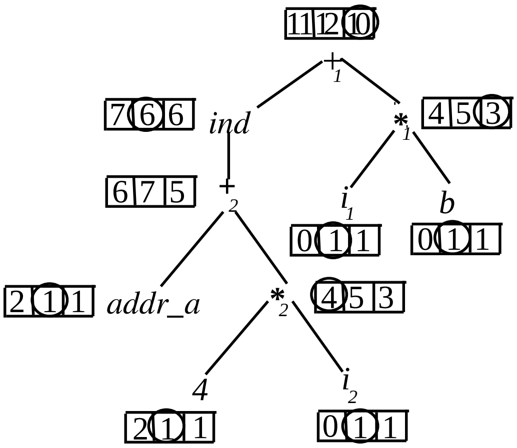


Figure 26: The earlier expression tree with costs calculated

$MOVE\ \#4,\ r_1$ **(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)**

4.5 Proof of optimality

We now present the result which shows that our theorem is optimal. Instead of a formal proof we shall provide the intuition behind the proof.

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.**

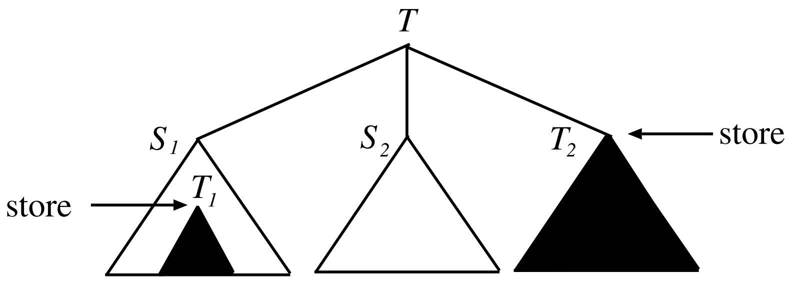


Figure 27: Outline of expression tree to illustrate optimality of Aho-Johnson algorithm

We shall try to illustrate the proof with an example. We only consider the case when $j > 0$. Consider an optimal program $P_1J_1P_2J_2PI$ 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_3P_4. For instance, P_3 could be the program computing the portion of S_1 in figure 27 which is not shaded, using j registers, and P_4 could be computing S_2 using $j-1$ registers. Also P_1J_1 and P_2J_2 must be computing the shaded subtrees T_1 and T_2. Now let us calculate the cost of this program.

- **$P_1J_1P_3$ 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 $P_1J_1P_3$ 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_2J_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.

We now estimate the complexity of the algorithm. The time required by Pass 1 is an, where a is a constant depending (i) linearly on the size of the instruction set, (ii) exponentially on the arity of the machine, i.e the average number of arguments of an instruction, and (iii) linearly on the number of registers in the machine. n is the number of nodes in the expression tree. The time required by Passes 2 and 3 is proportional to n. p Therefore the complexity of the algorithm is $O(n)$.

5 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.

6 References

1. Aho, A.V. and S.C. Johnson (1974): 'Optimal Code Generation for Expression Trees', *Journal of the Association for Computing Machinery*, Vol. 23, No. 3, pp 488-501.

2. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.

3. Barrett, W.A. and J.D. Couch (1977): *Compiler Construction*, Science Research Associates, Pennsylvania.

4. Bauer, F.L. and J. Eickel (1974): *Compiler Construction : An Advanced Course*, Springer-Verlag, Berlin.

5. Dhamdhere, D.M. (1983): *Compiler Construction – Principles and Practice*, Macmillan India, New Delhi.

6. Fischer, C.N. and R.J. LeBlanc (1988) : *Crafting a compiler*, Benjamin/Cummings, Menlo Park, California.

7. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.

8. Holub, A.I. (1993): *Compiler Design in C*, Prentice-Hall, Englewood-Cliffs.

9. Levine, J.R., T. Mason and D. Brown (1990): *Lex and Yacc*, 2^{nd} edition, O'Reilly & Associates, Sebastopol.

10. Lewis, P.M., D.J. Rosenkrantz, and R.E. Stearns (1976): *Compiler Design Theory*, Addison-Wesley, Reading.

11. Sethi, R. and J.D. Ullman (1970): 'The generation of Optimal Code for Arithmetic Expressions', *Journal of the Association for Computing Machinery*, Vol. 17, No. 4, pp 715-728.

12. Schreiner, A.T., H.G. Friedman Jr. (1985): *Introduction to Compiler Construction with UNIX* Prentice-Hall, Englewood Cliffs.

13. Tremblay, J.P. and P.G. Sorenson (1985): *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York.

14. Watson, D. (1989): *High Level Languages and their Compilers*, Addison-Wesley, Reading.