COMPILER DESIGN

UNIT – IV

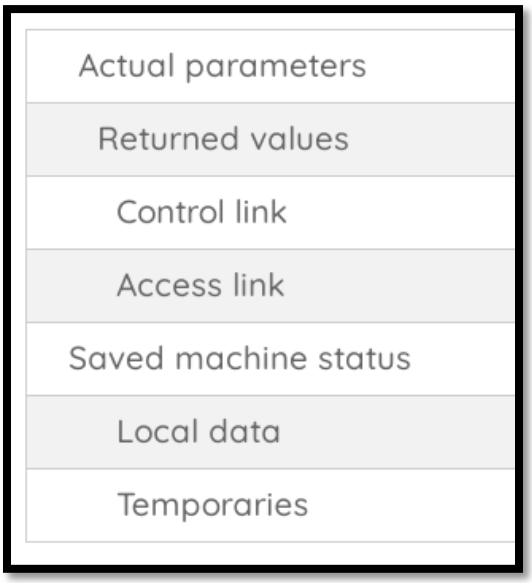
Stack Allocation of Space:

Almost all compilers for languages that use procedure, functions, or methods manage their run-time memory as a stack. Whenever a procedure is called, the local variable's space is pushed into a stack and popped off from the stack when the procedure terminates.

Activation Records

A run-time stack known as **a control stack** manages the procedure calls and returns. The **control stack** stores the activation record of each live activation. The top of the stack will hold the latest activation record.

The content of the activation record is given below. This record may vary according to the implemented languages.



Temporaries: The values which arise from the evaluation of expression will be held by temporaries.

Local data: The data belonging to the execution of the procedure is stored in local data.

Saved machine status: The status of the machine that might contain **register**, **program counter** before the call to the procedure is stored in saved machine status.

Access link: The data's information outside the local scope is stored in the access link**.**

Control link: The activation record of the caller is pointed by the control link.

Returned values: It represents the space for the return value of the called function if any.

Actual parameter: It represents the actual parameters used by the calling procedure.

Access to Nonlocal Data on the Stack:

- Data Access Without Nested Procedures
- Issues With Nested Procedures
- A Language With Nested Procedure Declarations
- Nesting Depth
- Access Links
- Manipulating Access Links
- Access Links for Procedure Parameters
- Displays

we consider how procedures access their data. Especially im-portant is the mechanism for finding data used within a procedure *p* but that does not belong to *p.* Access becomes more complicated in languages where procedures can be declared inside other procedures.

Heap Management:

Heap allocation is the most flexible allocation scheme. Allocation and deallocation of memory can be done at any time and any place depending upon the user's requirement. Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.

Heap management is specialized in data structure theory. There is generally some time and space overhead associated with heap manager. For efficiency reasons, it may be useful to handle small activation records of a particular size as a special case, as follows −

- For each size of interest, keep the linked list of free blocks of that size.
- If possible fill the request for size s with a block of size S', where S' is the smallest size greater than or equal to s. When the block is deallocated return back to the linked list.
- For a larger block of storage use the heap manager.

Properties of Heap Allocation

There are various properties of heap allocation which are as follows −

- **Space Efficiency**− A memory manager should minimize the total heap space needed by a program.

- **Program Efficiency**– A memory manager should make good use of the memory subsystem to allow programs to run faster. As the time taken to execute an instruction can vary widely depending on where objects are placed in memory.
- **Low Overhead**– Memory allocation and deallocation are frequent operations in many programs. These operations must be as efficient as possible. That is, it is required to minimize the overhead. The fraction of execution time spent performing allocation and deallocation.

Garbage collection:

Garbage collection (GC) is a dynamic approach to automatic memory management and heap allocation that processes and identifies dead memory blocks and reallocates storage for reuse. The primary purpose of garbage collection is to reduce memory leaks.

GC implementation requires three primary approaches, as follows:

- Mark-and-sweep - In process when memory runs out, the GC locates all accessible memory and then reclaims available memory.
- Reference counting - Allocated objects contain a reference count of the referencing number. When the memory count is zero, the object is garbage and is then destroyed. The freed memory returns to the memory heap.
- Copy collection - There are two memory partitions. If the first partition is full, the GC locates all accessible data structures and copies them to the second partition, compacting memory after GC process and allowing continuous free memory.

Introduction to Trace-Based Collection:

Trace-based collector whenever the free space is exhausted or its amount drops below some threshold. We begin this section by introducing the simplest "mark-and-sweep" gar-bage collection algorithm. We then describe the variety of trace-based algorithms in terms of four states that chunks of memory can be put in.

Introduction to Trace-Based Collection Introduction: Instead of collecting garbage as it is created, trace-based collectors run periodically to find unreachable objects and reclaim their space. Typically, we run the trace-based collector whenever the free space is exhausted or its amount drops below some threshold.

A trace-based garbage collector starts by labeling ("marking") all objects in the root set as "reachable," examines iteratively all the references in reachable objects to find more reachable objects, and labels them as such. This approach

must trace all the references before it can determine any object to be unreachable.

Tracing just-in-time compilation is a technique used by virtual machines to optimize the execution of a program at runtime. This is done by recording a linear sequence of frequently executed operations, compiling them to native machine code and executing them.

Issues in the Design of a Code Generator:

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

The following issue arises during the code generation phase:

1. **Input to code generator**
   The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.
2. **Target program**
   The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.
   - Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
   - Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
   - Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.
3. **Memory Management**
   Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three

address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. **Instruction selection**
   Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

   For example, the respective three-address statements would be translated into the latter code sequence as shown below:

   P:=Q+R

   S:=P+T

   MOV Q, R0

   ADD R, R0

   MOV R0, P

   MOV P, R0

   ADD T, R0

   MOV R0, S

   Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. **Register allocation issues**
   Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

   1. During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
   2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

   As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

   M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

Evaluation order

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

Approaches to code generation issues: Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

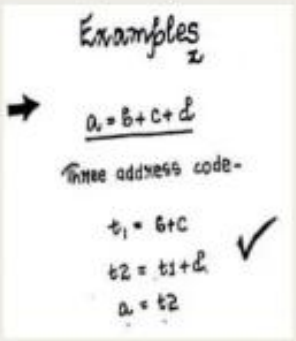
Basic Blocks and Flow Graphs:

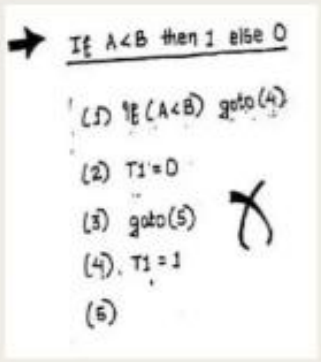
Basic Block

The basic block is a set of statements. The basic blocks do not have any in and out branches except entry and exit. It means the flow of control enters at the beginning and will leave at the end without any halt. The set of instructions of basic block executes in sequence.

- A *basic block* is a sequence of consecutive instructions which are always executed in sequence without halt or possibilty of branching.
- The basic block doesnot have any jump statements among them.

- When the first instruction is executed , all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control from the program.





Basic block Construction:

- *Input*: A sequence of three-address statements.
- *Output*: A list of basic blocks with each three-address statement in exactly one block.

1. **Determine the set of *leaders*,**
   a. The first statement is the leader.
   b. Any statement that is the target of a conditional or goto is a leader.
   c. Any statement that immediately follows conditional or goto is a leader.
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Consider the following three address code statements .Compute the basic blocks.

```
1)  PROD = 0
2)  I = 1
3)  T2=addr(A)-4
4)  T4 =addr(B)-4
5)  T1 =4*I
6)  T3= T2[T1]
7)  T5 = T4[T1]
8)  T6 =T3*T5
9)  PROD =PROD + T6
10) I =I+1
11) IF I<= 20 GOTO (5)
```

Solution:

- Because first statement is a leader ,so –

  PROD =0 is a leader.

- Because the target statement of conditional or unconditional goto statement is a leader, so-

  T1 =4*I is also a leader

So the given code can be portioned in to 2 blocks as :

B1:

```
PROD = 0
I = 1
T2=addr(A)-4
T4 =addr(B)-4
```

B2:

```
T1 =4*I
T3= T2[T1]
T5 = T4[T1]
T6 =T3*T5
PROD =PROD + T6
I =I+1
IF I<= 20 GOTO B2
```

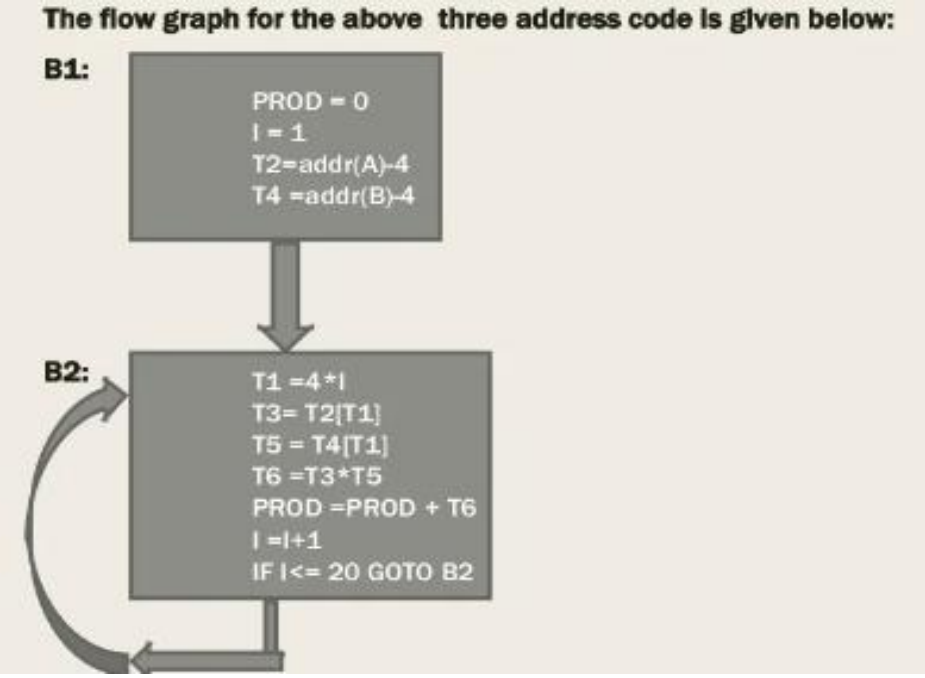
Flow Graph:

Introduction:

- A *flow graph* is a graphical representation of a sequence of instructions with control flow edges.
- A flow graph can be defined at the intermediate code level or target code level.
- The nodes of flow graphs are the basic blocks and flow-of-control to immediately follow node connected by directed arrow.

Points to remember:

- The basic blocks are the nodes to the flow graph .
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block B1 to B2 if B2 immediately follows B1 in the given sequence
- Then we say that B1 is the predecessor of B2.



Optimization of Basic Blocks:

Optimization is applied to the basic blocks after the intermediate code generation phase of the compiler. Optimization is the process of transforming a program that improves the code by consuming fewer resources and delivering high speed. In optimization, high-level codes are replaced by their equivalent efficient low-level codes. Optimization of basic blocks can be machine-dependent or machine-independent. These transformations are useful for improving the quality of code that will be ultimately generated from basic block.

There are two types of basic block optimizations:

1. Structure preserving transformations
2. Algebraic transformations

Structure-Preserving Transformations:

The structure-preserving transformation on basic blocks includes:

- Dead Code Elimination
- Common Subexpression Elimination
- Code movement

1. Dead Code Elimination:

Dead code is defined as that part of the code that never executes during the program execution. So, for optimization, such code or dead code is eliminated. The code which is never executed during the program (Dead code) takes time so, for optimization and speed, it is eliminated from the code. Eliminating the dead code increases the speed of the program as the compiler does not have to translate the dead code.

Example:

```
        // Program with Dead code
        int main()
        {
            x = 2
            if (x > 2)
              cout << "code"; // Dead code
            else
              cout << "Optimization";
            return 0;
        }
// Optimized Program without dead code
int main()
{
    x = 2;
    cout << "Optimization"; // Dead Code Eliminated
    return 0;
}
```

2. Common Subexpression Elimination:

The expression or sub-expression that has been appeared and computed before and appears again during the computation of the code is the common sub-expression.

In this technique, the sub-expression which are common are used frequently are calculated only once and reused when needed.

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Example:

Before elimination

```
a = 10;
b = a + 1 * 2;
c = a + 1 * 2;
//'c' has common expression as 'b'
d = c + a;
```

After elimination

```
a = 10;
b = a + 1 * 2;
d = b + a;
```

3. Code movement:

Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Example: before

```
for ( int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
```

After

```
x = y + z ;
for ( int j = 0 ; j < n ; j ++)
{
a[j] = 6 x j;
}
```

Algebraic Transformation:

Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Some of the algebraic transformation on basic blocks includes:

- Constant Folding
- Copy Propagation
- Strength Reduction

1. Constant Folding:

Solve the constant terms which are continuous so that compiler does not need to solve this expression.

- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

Example:

x = 2 * 3 + y ⇒ x = 6 + y (Optimized code)

2. Copy Propagation:

Copy propagation is used to replace the occurrence of target variables that are the direct assignments with their values.

A direct assignment is an instruction of the form **x = y**, which simply assigns the value of **y** to **x**.

It is of two types, Variable Propagation, and Constant Propagation.

Variable Propagation:

x = y ⇒ *z = y + 2 (Optimized code)*

z = x + 2

Constant Propagation:

x = 3 ⇒ *z = 3 + a (Optimized code)*

z = x + a

3. Strength Reduction:

Replace expensive statement/ instruction with cheaper ones.

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example:

x = 2 * y (costly) ⇒ x = y + y (cheaper)

x = 2 * y (costly) ⇒ x = y << 1 (cheaper)

A Simple Code Generator:

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

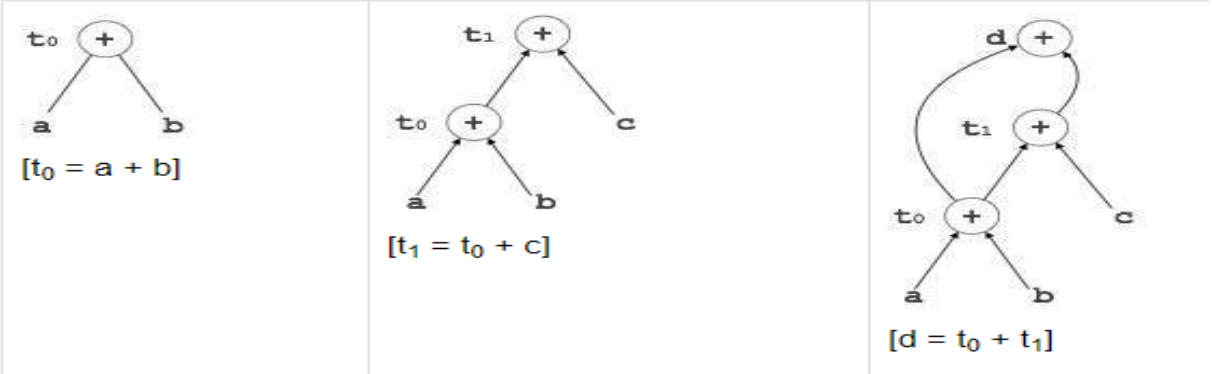
Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

```
t0 = a + b
t1 = t0 + c
d = t0 + t1
```



Peephole Optimization:

Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.

The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.

It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
   {
   int y, z;
   y = 10;
   z = x + y;
   return z;
   }
```

```
int add_ten(int x)
   {
   int y;
   y = 10;
   y = x + y;
   return y;
   }
```

```
int add_ten(int x)
   {
   int y = 10;
   return x + y;
   }
```

```
int add_ten(int x)
   {
   return x
   }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
  return x + 10;
  printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Register Allocation and Assignment:

Register are the fastest locations in the memory hierarchy. But unfortunately, this resource is limited. It comes under the most constrained resources of the target processor. Register allocation is an NP-complete problem. However, this problem can be reduced to graph coloring to achieve allocation and assignment. Therefore a good register allocator computes an effective approximate solution to a hard problem.

Allocation vs Assignment:

Allocation –

Maps an unlimited namespace onto that register set of the target machine.

- **Reg. to Reg. Model:** Maps virtual registers to physical registers but spills excess amount to memory.
- **Mem. to Mem. Model:** Maps some subset of the memory location to a set of names that models the physical register set.

Allocation ensures that code will fit the target machine's reg. set at each instruction.

Assignment –

Maps an allocated name set to the physical register set of the target machine.

- Assumes allocation has been done so that code will fit into the set of physical registers.
- No more than **'k'** values are designated into the registers, where 'k' is the no. of physical registers.

General register allocation is an NP-complete problem:

- Solved in polynomial time, when (no. of required registers) <= (no. of available physical registers).
- An assignment can be produced in linear time using Interval-Graph Coloring.

Local Register Allocation And Assignment:

Allocation just inside a basic block is called Local Reg. Allocation. Two approaches for local reg. allocation: Top-down approach and bottom-up approach.

Top-Down Approach is a simple approach based on 'Frequency Count'. Identify the values which should be kept in registers and which should be kept in memory.

Algorithm:

1. Compute a priority for each virtual register.
2. Sort the registers into priority order.
3. Assign registers in priority order.
4. Rewrite the code.

Moving beyond single Blocks:

- More complicated because the control flow enters the picture.
- Liveness and Live Ranges: Live ranges consist of a set of definitions and uses that are related to each other as they i.e. no single register can be common in a such couple of instruction/data.

Following is a way to find out Live ranges in a block. A live range is represented as an interval [i,j], where i is the definition and j is the last use.

Global Register Allocation and Assignment:

1. The main issue of a register allocator is minimizing the impact of spill code;

- Execution time for spill code.
- Code space for spill operation.
- Data space for spilled values.

2. Global allocation can't guarantee an optimal solution for the execution time of spill code.

3. Prime differences between Local and Global Allocation:

- The structure of a global live range is naturally more complex than the local one.
- Within a global live range, distinct references may execute a different number of times. (When basic blocks form a loop)

Dynamic Programming Code-Generation:

A dynamic programming algorithm is used to extend the class of machines for which optimal code can be generated from expressions trees in linear time. This algorithm works for a broad class of register machines with complex instruction sets.

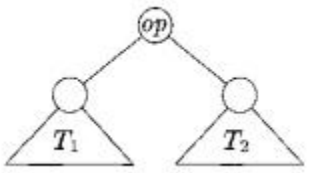
Such an algorithm is used to generate code for machines with *r* interchangeable registers *R0, R1, ..., Rr-1* and load, store and operation instructions.

Throughout this article, we will assume that every instruction has a cost of one unit however a dynamic programming algorithm works even with instructions having their own costs.

Contiguous Evaluation:

The dynamic programming algorithm partitions the problem of generating optimal code for an expression into the subproblems of generating optimal code for the subexpressions of the given expression. As a simple example, consider an expression *E* of the form *E1* + E_2. An optimal program for *E* is formed by combining optimal programs for *E1* and E_2, in one or the other order, followed by code to evaluate the operator +. The subproblems of generating optimal code for *E1* and E_2 are solved similarly.

An optimal program produced by the dynamic programming algorithm has an important property. It evaluates an expression $E = E1$ op E_2 "contigu-ously." We can appreciate what this means by looking at the syntax tree T for E:



Here Ti and T_2 are trees for $E1$ and E_2, respectively.

We say a program P evaluates a tree T *contiguously* if it first evaluates those subtrees of T that need to be computed into memory. Then, it evaluates the remainder of T either in the order Ti, T_2, and then the root, or in the order T_2, Ti, and then the root, in either case using the previously computed values from memory whenever necessary. As an example of noncontiguous evaluation, P might first evaluate part of Ti leaving the value in a register (instead of memory), next evaluate T_2, and then return to evaluate the rest of Ti.

For the register machine in this section, we can prove that given any machine-language program P to evaluate an expression tree T, we can find an equivalent program P' such that

P' is of no higher cost than P,

P' uses no more registers than P, and

P' evaluates the tree contiguously.

This result implies that every expression tree can be evaluated optimally by a contiguous program.