

TYPE CHECKING

TYPE CHECKING

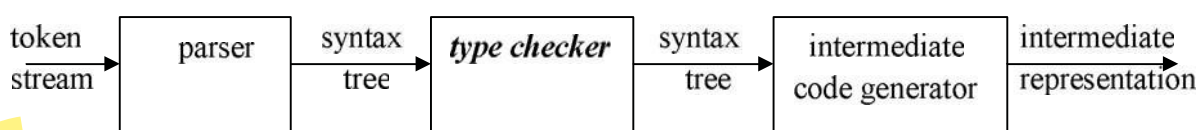
A compiler must check that the source program follows both syntactic and semantic conventions of the source language.

This checking, called *static checking*, detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.
2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as `break`, does not exist in switch statement.

Position of type checker



- A **type checker** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator `mod` in Pascal requires integer operands, so a type checker verifies that the operands of `mod` have type integer.
- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : “ if both operands of the arithmetic operators of `+`, `-` and `*` are of type integer, then the result is of type integer ”

Type Expressions

- The type of a language construct will be denoted by a “type expression.”
- A type expression is either a basic type or is formed by applying an operator called a **type constructor** to other type expressions.
- The sets of basic types and constructors depend on the language to be checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean*, *char*, *integer*, *real* are type expressions.

A special basic type, *type_error*, will signal an error during type checking; *void* denoting “the absence of a value” allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression.

Constructors include:

Arrays : If T is a type expression then $array(I, T)$ is a type expression denoting the type of an array with elements of type T and index set I .

Products : If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression.

Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

```
type row = record
    address: integer;
    lexeme: array[1..15] of char
end;

var table: array[1..101] of row;
```

declares the type name *row* representing the type expression *record((address X integer) X (lexeme X array(1..15,char)))* and the variable *table* to be an array of records of this type.

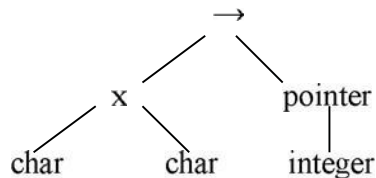
Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type “pointer to an object of type T”.

For example, *var p: ↑ row* declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression $D \rightarrow R$.

4. Type expressions may contain variables whose values are type expressions.

Tree representation for $\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$



Type systems

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic.
- Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

$P \rightarrow D ; E$
 $D \rightarrow D ; D \mid id : T$
 $T \rightarrow char \mid integer \mid array [num] of T \mid \uparrow T$
 $E \rightarrow literal \mid num \mid id \mid E mod E \mid E [E] \mid E \uparrow$

Translation scheme:

$P \rightarrow D ; E$
 $D \rightarrow D ; D$
 $D \rightarrow id : T \quad \{ addtype (id.entry, T.type) \}$
 $T \rightarrow char \quad \{ T.type := char \}$
 $T \rightarrow integer \quad \{ T.type := integer \}$
 $T \rightarrow \uparrow T_1 \quad \{ T.type := pointer(T_1.type) \}$
 $T \rightarrow array [num] of T_1 \quad \{ T.type := array (1 \dots num.val, T_1.type) \}$

In the above language,

- There are two basic types : char and integer ;
- *type_error* is used to signal errors;
- the prefix operator \uparrow builds a pointer type. Example , $\uparrow integer$ leads to the type expression **pointer (integer)**.

Type checking of expressions

In the following rules, the attribute *type* for *E* gives the type expression assigned to the expression generated by *E*.

1. $E \rightarrow \text{literal}$ $\{ E.type := char \}$
 $E \rightarrow \text{num}$ $\{ E.type := integer \}$

Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. $E \rightarrow \text{id}$ $\{ E.type := lookup (\text{id.entry}) \}$
lookup (e) is used to fetch the type saved in the symbol table entry pointed to by *e*.

3. $E \rightarrow E_1 \text{ mod } E_2$ $\{ E.type := \text{if } E_1.type = integer \text{ and } E_2.type = integer \text{ then } integer \text{ else } type_error \}$

The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. $E \rightarrow E_1 [E_2]$ $\{ E.type := \text{if } E_2.type = integer \text{ and } E_1.type = array(s,t) \text{ then } t \text{ else } type_error \}$

In an array reference $E_1 [E_2]$, the index expression E_2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E_1 .

5. $E \rightarrow E_1 \uparrow$ $\{ E.type := \text{if } E_1.type = pointer(t) \text{ then } t \text{ else } type_error \}$

The postfix operator \uparrow yields the object pointed to by its operand. The type of $E \uparrow$ is the type *t* of the object pointed to by the pointer E .

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

$S \rightarrow \text{id} := E$ $\{ S.type := \text{if id.type} = E.type \text{ then } void \text{ else } type_error \}$

2. Conditional statement:

$S \rightarrow \text{if } E \text{ then } S_1$ $\{ S.type := \text{if } E.type = boolean \text{ then } S_1.type \text{ else } type_error \}$

3. While statement:

$S \rightarrow \text{while } E \text{ do } S_1$ $\{ S.type := \text{if } E.type = boolean \text{ then } S_1.type \text{ else } type_error \}$

4. Sequence of statements:

$$S \rightarrow S_1 ; S_2 \quad \{ S.type := \text{if } S_1.type = \text{void} \text{ and } S_1.type = \text{void} \text{ then void} \\ \text{else type_error} \}$$

Type checking of functions

The rule for checking the type of a function application is :

$$E \rightarrow E_1 (E_2) \quad \{ E.type := \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \\ \text{else type_error} \}$$

MODULE 4 - RUN-TIME ENVIRONMENTS

SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.

For example, the following is the definition of procedure named *readarray* :

```
procedure readarray;  
  var i : integer;  
  begin  
    for i := 1 to 9 do read(a[i])  
  end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name.

Declarations may be explicit, such as:

```
var i : integer ;
```

or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

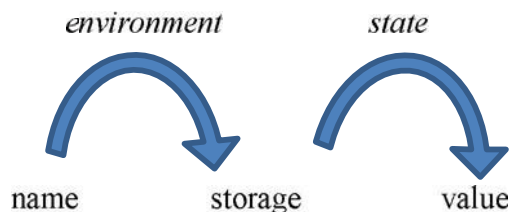
The portion of the program to which a declaration applies is called the *scope* of that declaration.

Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. “Data object” corresponds to a storage location that holds values.

The term *environment* refers to a function that maps a name to a storage location.

The term *state* refers to a function that maps a storage location to the value held there.

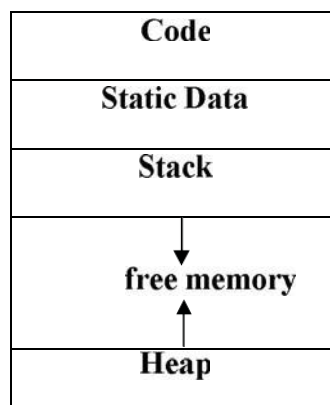


When an *environment* associates storage location s with a name x , we say that x is *bound* to s . This association is referred to as a *binding* of x .

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

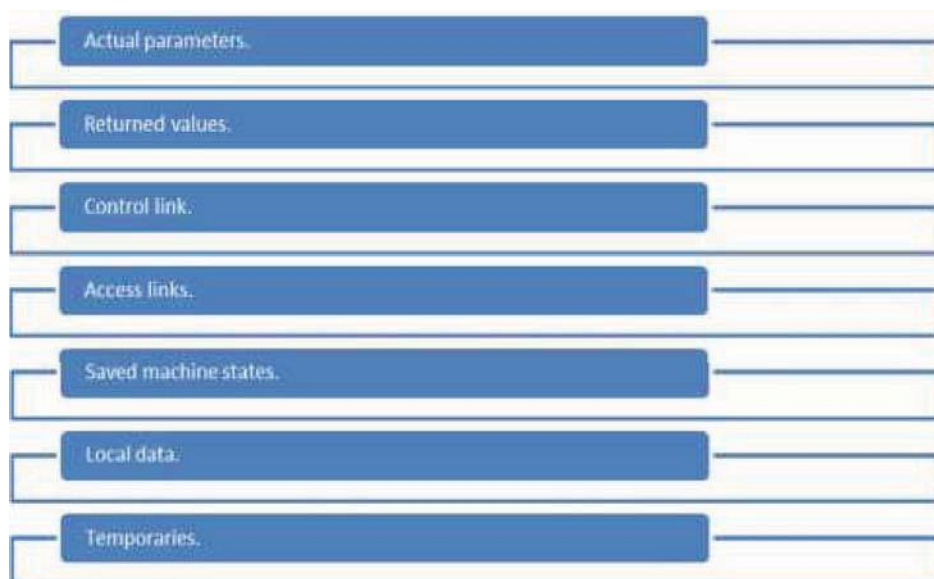
Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack*.
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

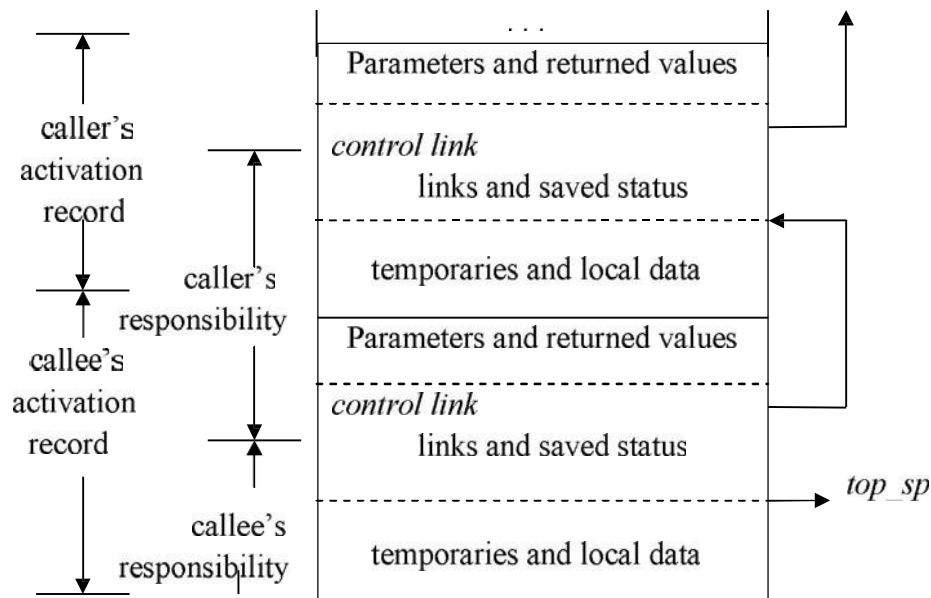
STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
 - Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

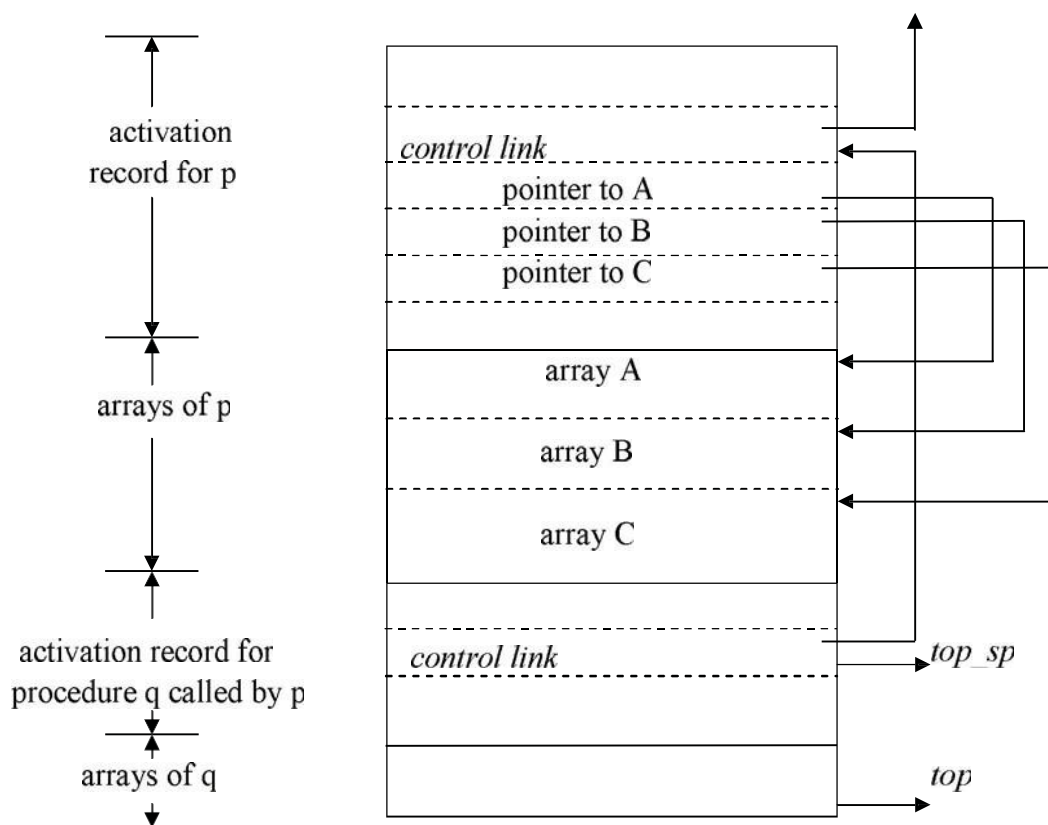


Division of tasks between caller and callee

- The calling sequence and its division between caller and callee are as follows.
 - The caller evaluates the actual parameters.
 - The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
 - The callee saves the register values and other status information.
 - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:
 - The callee places the return value next to the parameters.
 - Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
 - Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



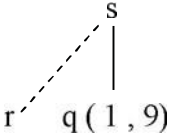
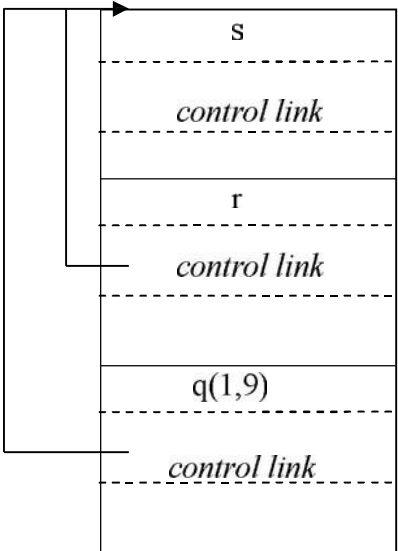
Access to dynamically allocated arrays

- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
 2. A called activation outlives the caller.
- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
 - Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

MODULE-4 INTERMEDIATE CODE GENERATION

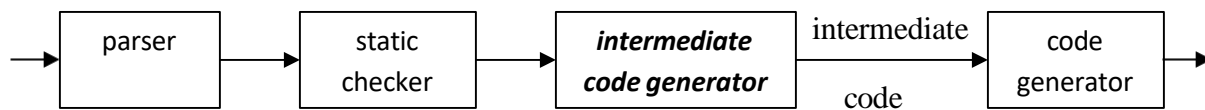
INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

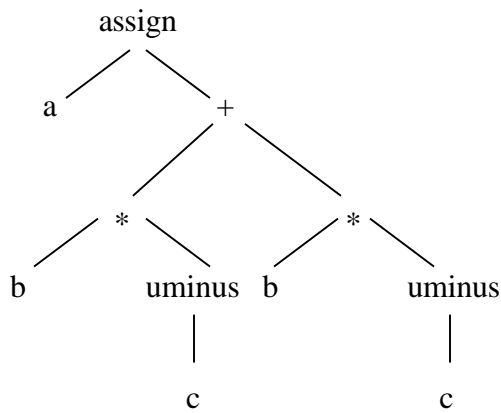
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

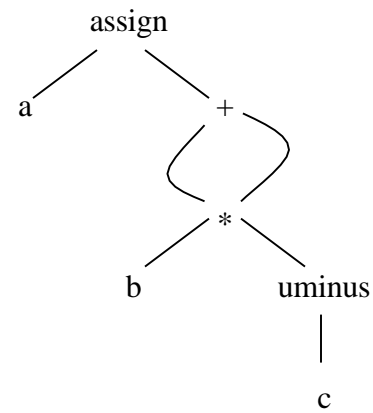
Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A **dag** (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a := b * - c + b * - c** are as follows:



(a) Syntax tree



(b) Dag

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Three-Address Code:

Three-address code is a sequence of statements of the general form $x :$

$= y \text{ op } z$

where x , y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like $x + y * z$ might be translated into a sequence

$t_1 := y * z$
 $t_2 := x + t_1$

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

$t_1 := -c$

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

$t_2 := b * t_1$

$t_5 := t_2 + t_2$

$a := t_5$

(a) Code for the syntax tree

(b) Code for the dag

The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form $x := y \text{ op } z$, where op is a binary arithmetic or logical operation.
2. Assignment instructions of the form $x := \text{op } y$, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. Copy statements of the form $x := y$ where the value of y is assigned to x .
4. The unconditional jump **goto L**. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as **if $x \text{ relop } y$ goto L**. This instruction applies a relational operator ($<, =, >=$, etc.) to x and y , and executes the statement with label L next if x stands in relation relop to y . If not, the three-address statement following **if $x \text{ relop } y$ goto L** is executed next, as in the usual sequence.
6. *param* x and *call* p, n for procedure calls and *return* y , where y representing a returned value is optional. For example,
 param x_1
 param x_2
 ...
 param x_n
 call p, n
generated as part of a call of the procedure $p(x_1, x_2, \dots, x_n)$.
7. Indexed assignments of the form $x := y[i]$ and $x[i] := y$.
8. Address and pointer assignments of the form $x := \&y$, $x := *y$, and $*x := y$.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are, *op*, *arg1*, *arg2* and *result*.
- The *op* field contains an internal code for the operator. The three-address statement $x := y \text{ op } z$ is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result*.
- The contents of fields *arg1*, *arg2* and *result* are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op*, *arg1* and *arg2*.
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples*.

	<i>op</i>	<i>arg1</i>	<i>arg2</i>	<i>result</i>
(0)	uminus	c		t ₁
(1)	*	b	t ₁	t ₂
(2)	uminus	c		t ₃
(3)	*	b	t ₃	t ₄
(4)	+	t ₂	t ₄	t ₅
(5)	:=	t ₃		a

(a) Quadruples

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like $x[i] := y$ requires two entries in the triple structure as shown as below while $x := y[i]$ is naturally represented as two operations.

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	[] =	x	i
(1)	assign	(0)	y

(a) $x[i] := y$

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	= []	y	i
(1)	assign	x	(0)

(b) $x := y[i]$

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	<i>statement</i>
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w , then the i th element of array A begins in location

$$base + (i - low) \times w$$

where low is the lower bound on the subscript and $base$ is the relative address of the storage allocated for the array. That is, $base$ is the relative address of $A[low]$.

The expression can be partially evaluated at compile time if it is rewritten as

$$i \times w + (base - low \times w)$$

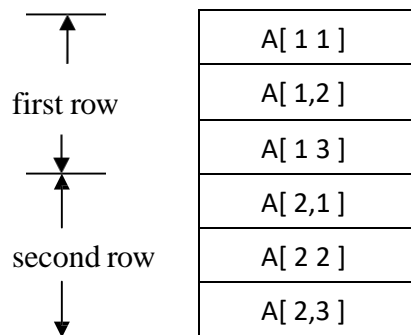
The subexpression $c = base - low \times w$ can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of $A[i]$ is obtained by simply adding $i \times w$ to c .

Address calculation of multi-dimensional arrays:

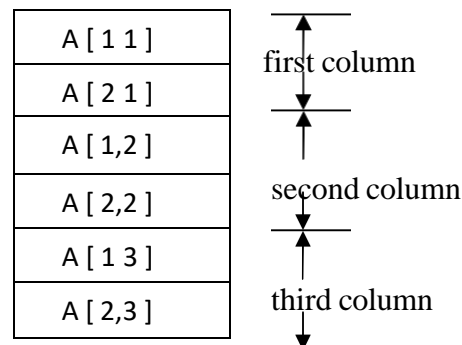
A two-dimensional array is stored in of the two forms :

- Row-major (row-by-row)
- Column-major (column-by-column)

Layouts for a 2 x 3 array



(a) ROW-MAJOR



(b) COLUMN-MAJOR

In the case of row-major form, the relative address of $A[i_1, i_2]$ can be calculated by the formula

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

where, low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if $high_2$ is the upper bound on the value of i_2 , then $n_2 = high_2 - low_2 + 1$.

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((i_1 \times n_2) + i_2) \times w + (base - (low_1 \times n_2 + low_2) \times w)$$

Generalized formula:

The expression generalizes to the following expression for the relative address of $A[i_1, i_2, \dots, i_k]$

$$((\dots ((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w + base - ((\dots ((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k) \times w$$

for all j , $n_j = high_j - low_j + 1$

BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and**, **or**, and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form $E_1 \text{ relop } E_2$, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$$

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false *numerically* and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by *flow of control*, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for

$a \text{ or } b \text{ and not } c$

is the three-address sequence

$t_1 := \text{not } c$

$t_2 := b \text{ and } t_1$

$t_3 := a \text{ or } t_2$

- A relational expression such as $a < b$ is equivalent to the conditional statement
if $a < b$ then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :   if a < b goto 103
101 :   t := 0
102 :   goto 104
103 :   t := 1
104 :
```

Translation of $a < b$ or $c < d$ and $e < f$

```
100 : if a < b goto 103          107 : t2 := 1
101 : t1 := 0                    108 : if e < f goto 111
102 : goto 104                  109 : t3 := 0
103 : t1 := 1                    110 : goto 112
104 : if c < d goto 107          111 : t3 := 1
105 : t2 := 0                    112 : t4 := t2 and t3
106 : goto 108                  113 : t5 := t1 or t4
```

Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

```
S → if E then S1
   | if E then S1 else S2
   | while E do S1
```

In each of these productions, E is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- $E.true$ is the label to which control flows if E is true, and $E.false$ is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation $S.code$ to the three-address instruction immediately following $S.code$.
- $S.next$ is a label that is attached to the first three-address instruction to be executed after the code for S .

CASE STATEMENTS

The “switch” or “case” statement is available in a variety of languages. The switch-statement syntax is as shown below :

Switch-statement syntax

```
switch expression
  begin
    case value : statement
    case value : statement
    . . .
    case value : statement
    default : statement
  end
```

There is a selector expression, which is to be evaluated, followed by n constant values that the expression might take, including a default “value” which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression.
3. Execute the statement associated with the value found.

Step (2) can be implemented in one of several ways :

- By a sequence of conditional **goto** statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
- If the number of cases is large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n -way branch exists. If the values all lie in some small range, say i_{\min} to i_{\max} , and the number of different values is a reasonable fraction of $i_{\max} - i_{\min}$, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset $j - i_{\min}$ and the label for the default in entries not filled otherwise. To perform switch,

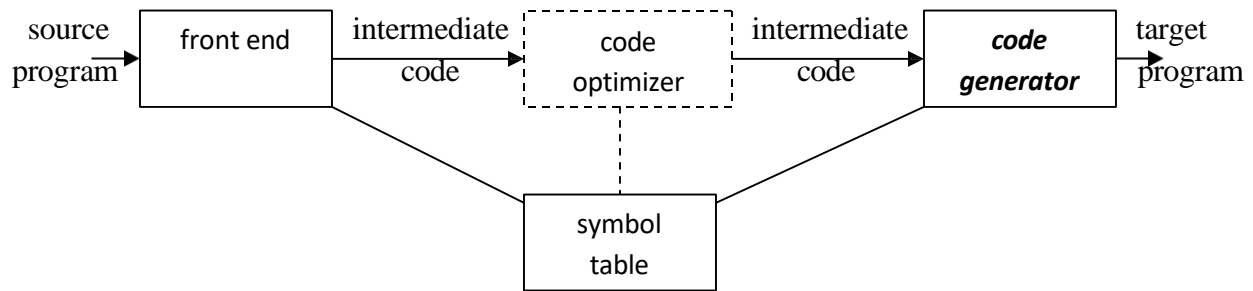
evaluate the expression to obtain the value of j , check the value is within range and transfer to the table entry at offset $j-i_{\min}$.



MODULE-4 CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representation can be :
 - a. Linear representation such as postfix notation
 - b. Three address representation such as quadruples
 - c. Virtual machine representation such as stack machine code
 - d. Graphical representations such as syntax trees and dags.
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :
 - a. Absolute machine language
 - It can be placed in a fixed memory location and can be executed immediately.

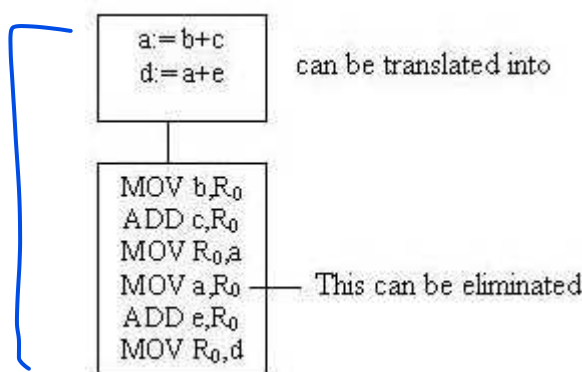
- b. Relocatable machine language
 - It allows subprograms to be compiled separately.
- c. Assembly language
 - Code generation is made easier.

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions. For example,
 - j : **goto** i generates jump instruction as follows :
 - if $i < j$, a backward jump instruction with target address equal to location of code for quadruple i is generated.
 - if $i > j$, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j . When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :
 - **Register allocation** – the set of variables that will reside in registers at a point in the program is selected.

- **Register assignment** – the specific register that a variable will reside in is picked.
- Certain machine requires even-odd *register pairs* for some operands and results. For example, consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair

y – divisor

even register holds the remainder

odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

- A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:
 - t₁ := a * a
 - t₂ := a * b
 - t₃ := 2 * t₂
 - t₄ := t₁ + t₃
 - t₅ := b * b
 - t₆ := t₄ + t₅

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
 - a. The first statement is a leader.
 - b. Any statement that is the target of a conditional or unconditional goto is a leader.
 - c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

- Consider the following source code for dot product of two vectors a and b of length 20

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i + 1;
    end
    while i <= 20
end
```

- The three-address code for the above source program is given as :

(1)	prod := 0
(2)	i := 1
(3)	t ₁ := 4 * i
(4)	t ₂ := a[t ₁] /*compute a[i] */
(5)	t ₃ := 4 * i
(6)	t ₄ := b[t ₃] /*compute b[i] */

(7)	$t_5 := t_2 * t_4$
(8)	$t_6 := \text{prod} + t_5$
(9)	$\text{prod} := t_6$
(10)	$t_7 := i + 1$
(11)	$i := t_7$
(12)	if $i \leq 20$ goto (3)

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations
- Algebraic transformations

1. Structure-preserving transformations:

a) Common subexpression elimination:

$a := b + c$		$a := b + c$
$b := a - d$	\longrightarrow	$b := a - d$
$c := b + c$		$c := b + c$
$d := a - d$		$d := b$

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

b) ~~Dead~~-code elimination:

Suppose x is dead, that is, never subsequently used, at the point where the statement $x := y + z$ appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) ~~Renaming~~ temporary variables:

A statement $t := b + c$ (t is a temporary) can be changed to $u := b + c$ (u is a new temporary) and all uses of this instance of t can be changed to u without changing the value of the basic block.

Such a block is called a *normal-form block*.

d) Interchange of statements:

Suppose a block has the following two adjacent statements:

$t1 := b + c$
 $t2 := x + y$

We can interchange the two statements without affecting the value of the block if and only if neither x nor y is $t1$ and neither b nor c is $t2$.

2. Algebraic transformations:

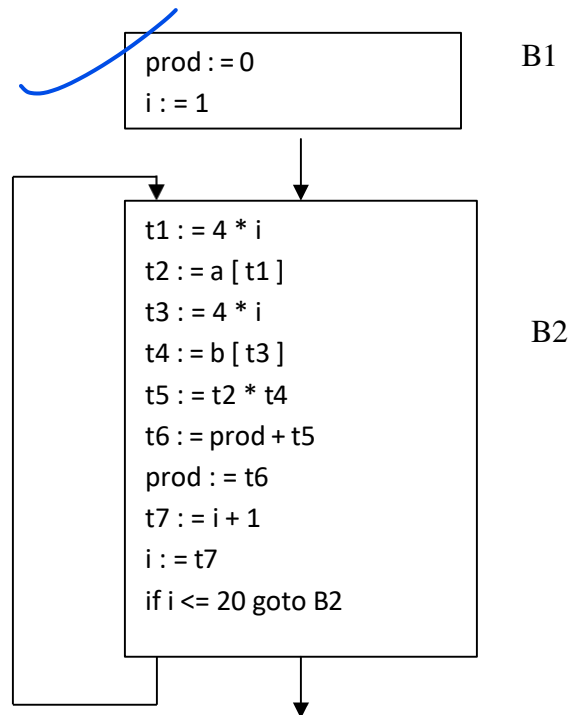
Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

- $x := x + 0$ or $x := x * 1$ can be eliminated from a basic block without changing the set of expressions it computes.
- The exponential statement $x := y * * 2$ can be replaced by $x := y * y$.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:



- B₁ is the *initial* node. B₂ immediately follows B₁, so there is an edge from B₁ to B₂. The target of jump from last statement of B₁ is the first statement B₂, so there is an edge from B₁ (last statement) to B₂ (first statement).
- B₁ is the *predecessor* of B₂, and B₂ is a *successor* of B₁.

Loops

- A loop is a collection of nodes in a flow graph such that
 1. All nodes in the collection are *strongly connected*.
 2. The collection of nodes has a unique *entry*.
- A loop that contains no other loops is called an inner loop.

NEXT-USE INFORMATION

- If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

Input: Basic block B of three-address statements

Output: At each statement $i: x = y \text{ op } z$, we attach to i the liveness and next-uses of x , y and z .

Method: We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveness of x , y and z .
2. In the symbol table, set x to “not live” and “no next use”.
3. In the symbol table, set y and z to “live”, and next-uses of y and z to i .

Symbol Table:

Names	Liveness	Next-use
x	not live	no next-use
y	Live	i
z	Live	i

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.
- For example: consider the three-address statement $a := b+c$
It can have the following sequence of codes:

ADD R_j, R_i Cost = 1 // if R_i contains b and R_j contains c

(or)

ADD c, R_i Cost = 2 // if c is in a memory location

(or)

MOV c, R_j Cost = 3 // move c from memory to R_j and add

ADD R_j, R_i

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form $x := y \text{ op } z$, perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation $y \text{ op } z$ should be stored.
2. Consult the address descriptor for y to determine y' , the current location of y . Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L , generate the instruction **MOV y' , L** to place a copy of y in L .
3. Generate the instruction **OP z' , L** where z' is a current location of z . Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L . If x is in L , update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of $x := y \text{ op } z$, those registers will no longer contain y or z .

Generating Code for Assignment Statements:

- The assignment $d := (a-b) + (a-c) + (a-c)$ might be translated into the following three-address code sequence:

$t := a - b$

$u := a - c$

$v := t + u$

$d := v + u$

with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
$t := a - b$	MOV a, R0 SUB b, R0	R0 contains t	t in R0
$u := a - c$	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
$v := t + u$	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
$d := v + u$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements

a := b [i] and **a [i] := b**

Statements	Code Generated	Cost
a := b[i]	MOV b(R _i), R	2
a[i] := b	MOV b, a(R _i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments

a := *p and ***p := a**

Statements	Code Generated	Cost
a := *p	MOV *R _p , a	2
*p := a	MOV a, *R _p	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x := y +z if x < 0 goto z	MOV y, R ₀ ADD z, R ₀ MOV R ₀ , x CJ< z

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
 1. Leaves are labeled by unique identifiers, either variable names or constants.
 2. Interior nodes are labeled by an operator symbol.
 3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) $x := y \text{ OP } z$

Case (ii) $x := \text{OP } y$

Case (iii) $x := y$

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

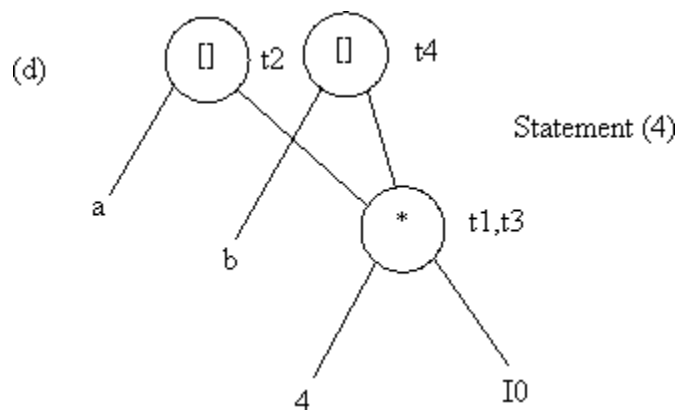
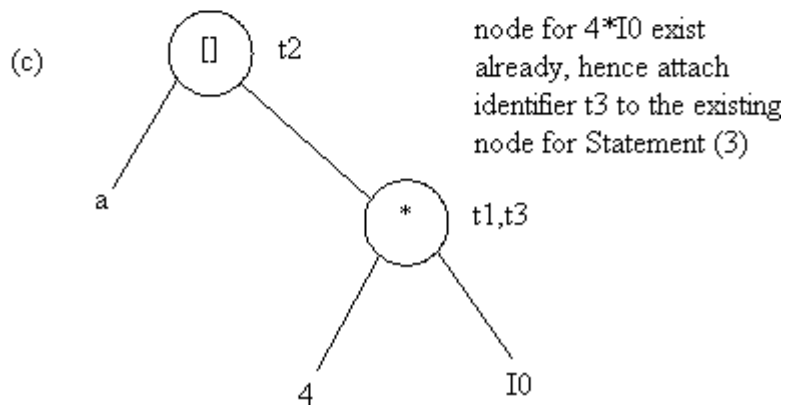
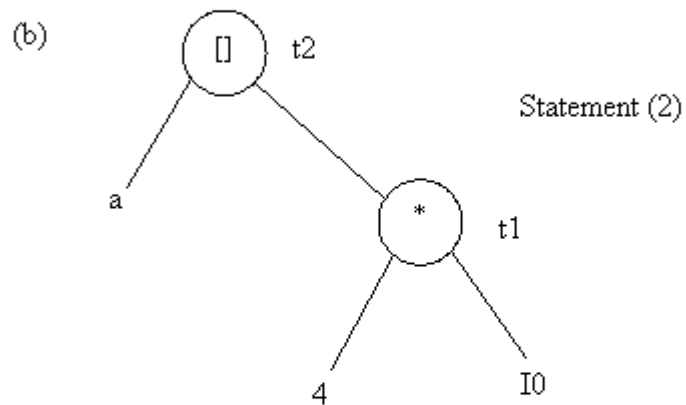
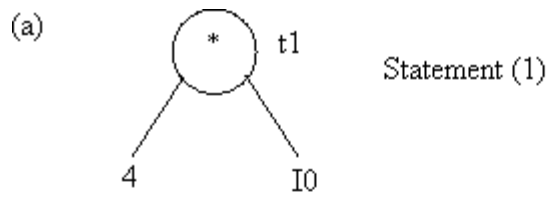
identifiers for the node n found in step 2 and set node(x) to n .

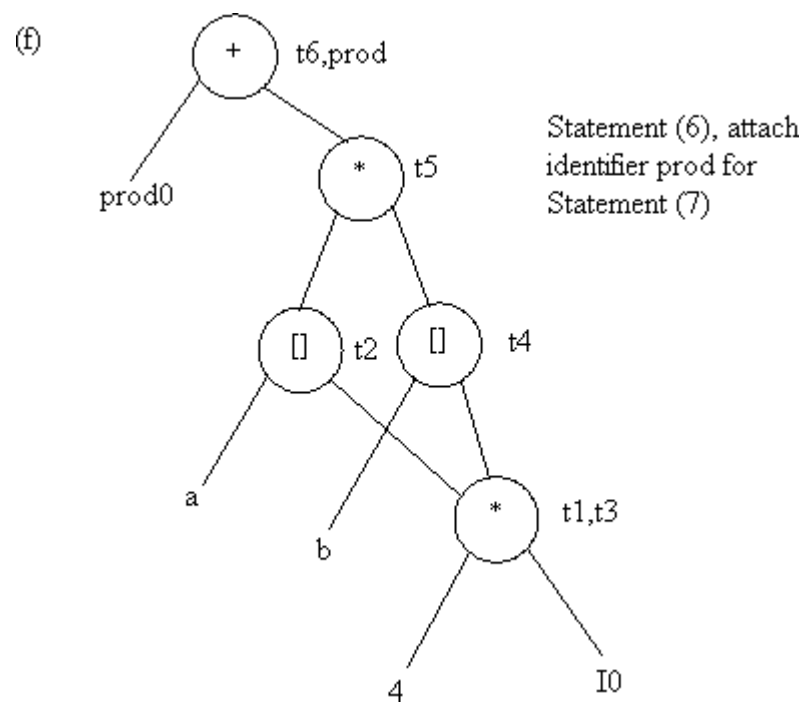
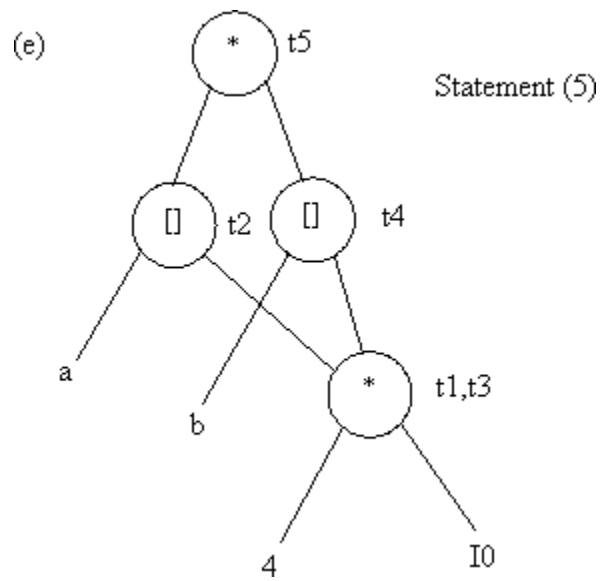
Example: Consider the block of three- address statements:

1. $t_1 := 4 * i$
2. $t_2 := a[t_1]$
3. $t_3 := 4 * i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2 * t_4$
6. $t_6 := \text{prod} + t_5$
7. $\text{prod} := t_6$
8. $t_7 := i + 1$
9. $i := t_7$
10. if $i \leq 20$ goto (1)

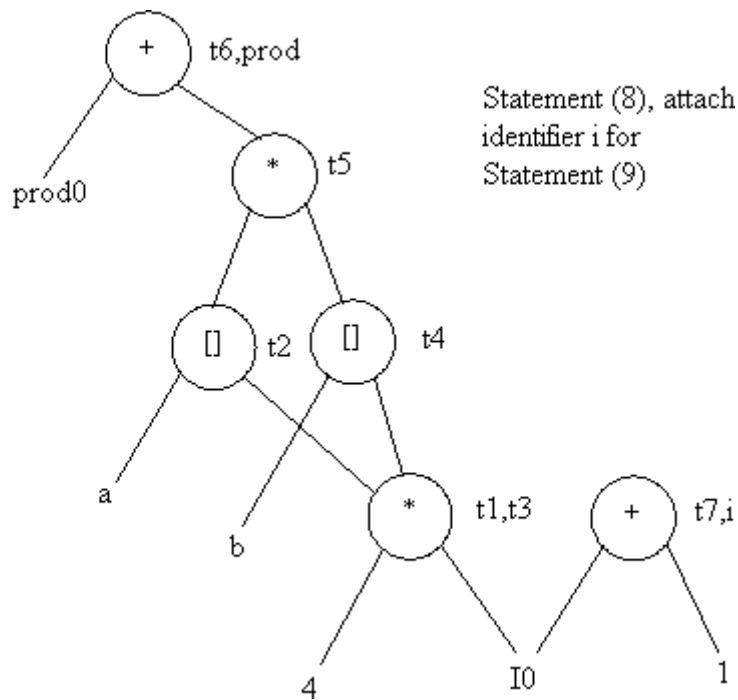


Stages in DAG Construction

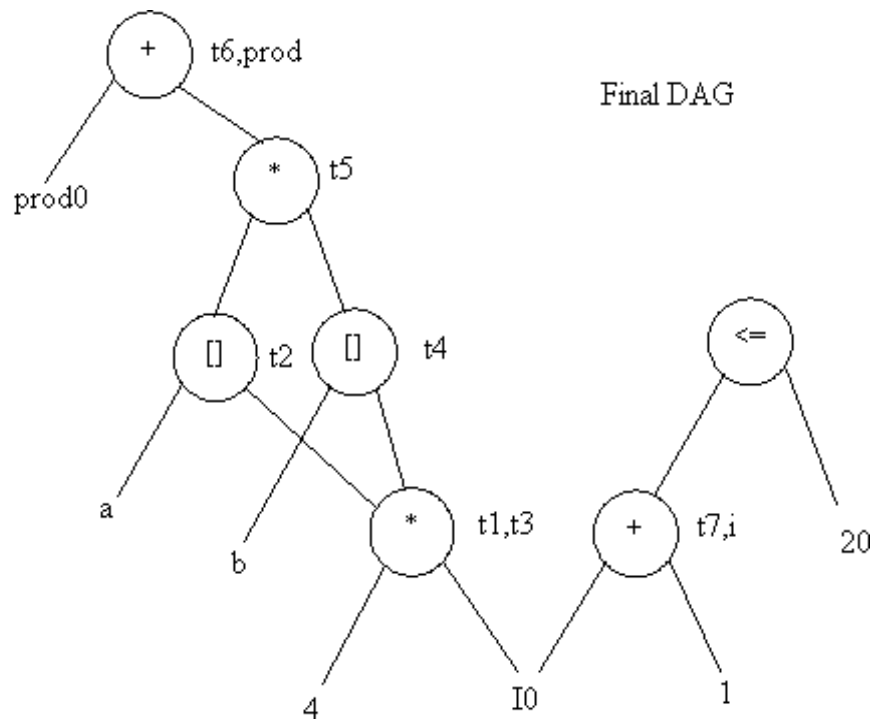




(g)



(h)



Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

GENERATING CODE FROM DAGs


The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
 $t_2 := c + d$
 $t_3 := e - t_2$
 $t_4 := t_1 - t_3$



Generated code sequence for basic block:

MOV a , R₀
ADD b , R₀
MOV c , R₁
ADD d , R₁
MOV R₀ , t₁
MOV e , R₀
SUB R₁ , R₀
MOV t₁ , R₁
SUB R₀ , R₁
MOV R₁ , t₄

Rearranged basic block:

Now t₁ occurs immediately before t₄.

$t_2 := c + d$
 $t_3 := e - t_2$
 $t_1 := a + b$
 $t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R₀
ADD d , R₀
MOV a , R₀
SUB R₀ , R₁
MOV a , R₀
ADD b , R₀
SUB R₁ , R₀
MOV R₀ , t₄

In this order, two instructions **MOV R₀ , t₁** and **MOV t₁ , R₁** have been saved.

MODULE - 4 - CODE OPTIMIZATION

INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.
- Optimizations are classified into two categories. They are
 - Machine independent optimizations:
 - Machine dependant optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

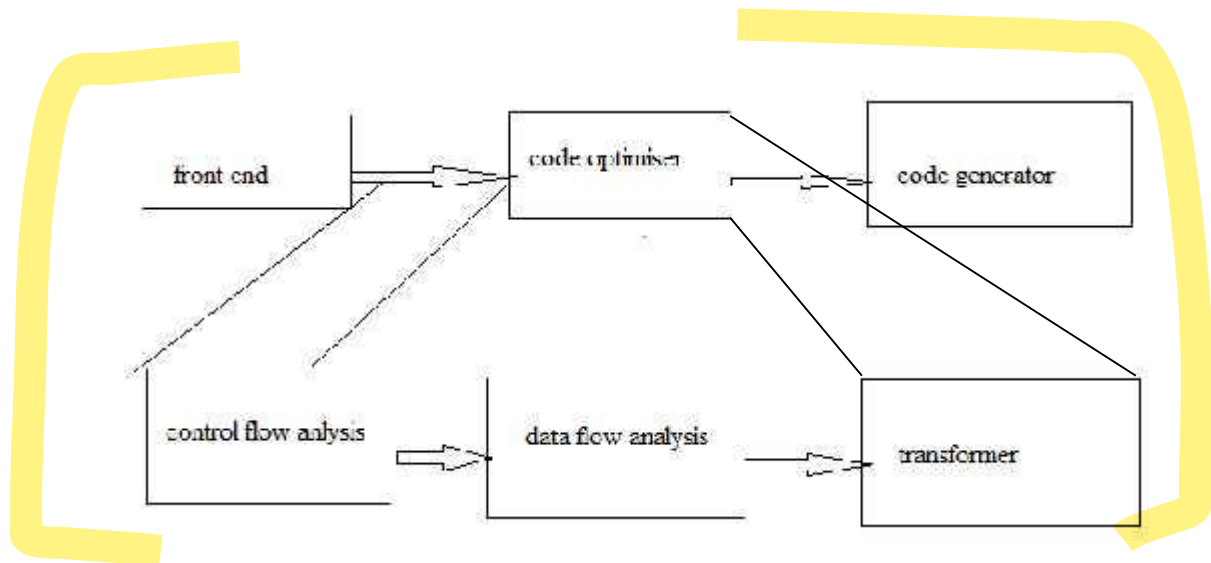
Machine dependant optimizations:

- Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.
- ✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.
- ✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an “optimization” may slow down a program slightly.
- ✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:



- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
 - control flow graph
 - Call graph
- Data flow analysis (DFA) is the process of ascertaining and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPAL SOURCES OF OPTIMISATION

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations
 - ✓ Common sub expression elimination,
 - ✓ Copy propagation,
 - ✓ Dead-code elimination, and
 - ✓ Constant folding

are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

➤ Common Sub expressions elimination:

- An occurrence of an expression **E** is called a **common sub-expression** if **E** was previously computed, and the values of variables in **E** have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
- For example

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

The above code can be **optimized using the common sub-expression elimination** as

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression **t4: =4*i** is **eliminated** as its computation is already in t₁. And value of i is not been changed from definition to use.

➤ Copy Propagation:

- Assignments of the form **f := g** called **copy statements**, or **copies** for short. The idea **behind the copy-propagation transformation** is to use **g** for **f**, whenever possible after the copy statement **f := g**. Copy propagation means use of **one variable instead of another**. This may not appear to be an improvement, but as we shall see it gives us an opportunity to **eliminate x**.
- For example:

```
x=Pi;
.....
A=x*r*r;
```

The optimization using **copy propagation** can be done as follows:

```
A=Pi*r*r;
```

Here the **variable x** is eliminated

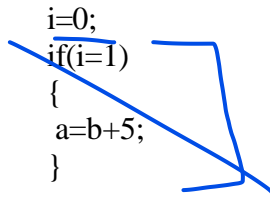
➤ Dead-Code Eliminations:

- A variable is **live** at a point in a program if its value can be used subsequently; otherwise, it is **dead** at that point. A related idea is **dead or useless code**, statements that compute

values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i=0;  
if(i=1)  
{  
  a=b+5;  
}
```



Here, 'if' statement is dead code because this condition will never get satisfied.

➤ Constant folding:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.
- ✓ For example,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

➤ Loop Optimizations:

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:
 - ✓ code motion, which moves code outside a loop;
 - ✓ Induction-variable elimination, which we apply to replace variables from inner loop.
 - ✓ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

➤ Code Motion:

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2)  /* statement does not change limit*/
```

Code motion will result in the equivalent of

```

t= limit-2;
while (i<=t)    /* statement does not change limit or t */

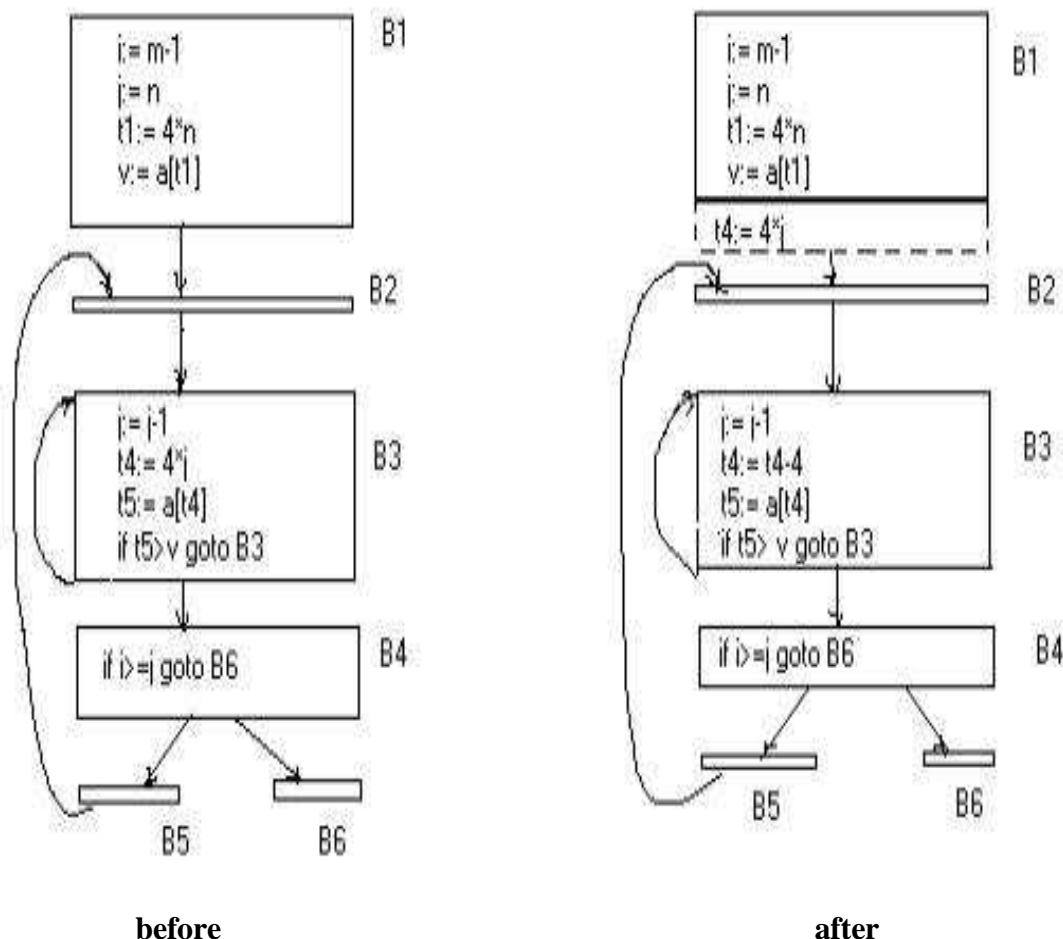
```

➤ Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because $4*j$ is assigned to t_4 . Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:

As the relationship $t_4 := 4*j$ surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement $j := j-1$ the relationship $t_4 := 4*j-4$ must hold. We may therefore replace the assignment $t_4 := 4*j$ by $t_4 := t_4-4$. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship $t_4 := 4*j$ on entry to the block B3, we place an initialization of t_4 at the end of the block where j itself is



initialized, shown by the dashed addition to block B1 in second Fig.

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

➤ **Reduction In Strength:**

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- ✓ Structure-Preserving Transformations
- ✓ Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- ✓ Common sub-expression elimination
- ✓ Dead code elimination
- ✓ Renaming of temporary variables
- ✓ Interchange of two independent adjacent statements.

➤ **Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

```
a: =b+c
b: =a-d
c: =b+c
d: =a-d
```

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

```
a: = b+c
b: = a-d
c: = a
d: = b
```

➤ **Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

➤ **Renaming of temporary variables:**

- A statement $t := b + c$ where t is a temporary name can be changed to $u := b + c$ where u is another temporary name, and change all uses of t to u .
- In this we can transform a basic block to its equivalent block called normal-form block.

➤ **Interchange of two independent adjacent statements:**

- Two statements

$t_1 := b + c$

$t_2 := x + y$

can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2 .

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression $2 * 3.14$ would be replaced by 6.28.
- The relational operators \leq , \geq , $<$, $>$, $+$ and $=$ sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

$a := b + c$
 $e := c + d + b$

the following intermediate code may be generated:

$a := b + c$
 $t := c + d$
 $e := t + b$

- Example:

$x := x + 0$ can be removed

$x := y ** 2$ can be replaced by a cheaper statement $x := y * y$

- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate $x*y-x*z$ as $x*(y-z)$ but it may not evaluate $a+(b-c)$ as $(a+b)-c$.

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:
 - ✓ Redundant-instructions elimination
 - ✓ Flow-of-control optimizations
 - ✓ Algebraic simplifications
 - ✓ Use of machine idioms
 - ✓ Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) **MOV R₀,a**

(2) **MOV a,R₀**

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R₀. If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0

....

If ( debug ) {

    Print debugging information

}
```

- In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2

goto L2

L1: print debugging information

L2: ..... (a)
```

- One obvious peephole optimization is to eliminate jumps over jumps. Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2

Print debugging information

L2: ..... (b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

If debug $\neq 0$ goto L2

Print debugging information

L2: (c)

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

....

L1: goto L2

by the sequence

goto L2

....

L1: goto L2

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump. Similarly, the sequence

if a < b goto L1

....

L1: goto L2

can be replaced by

If a < b goto L2

....

L1: goto L2

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

.....

L1: if a < b goto L2

L3: (1)

- May be replaced by

If a < b goto L2

goto L3

.....

L3: (2)

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1). Thus (2) is superior to (1) in execution time

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

$x := x + 0$

Or

$x := x * 1$

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X * X$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like $i := i + 1$.

$i := i + 1 \rightarrow i++$

$i := i - 1 \rightarrow i--$

✓ **s and delete statement j: =s.**