4th Unit

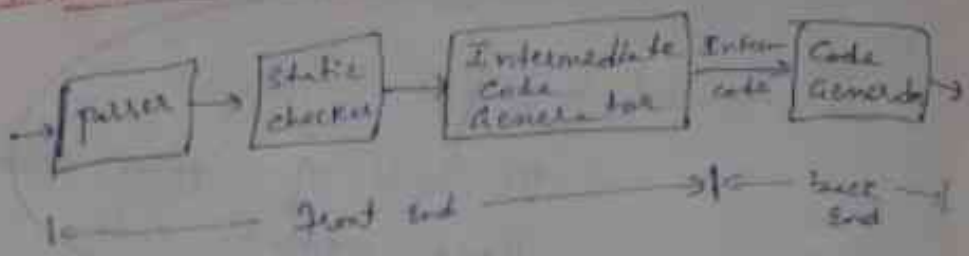
Intermediate Code Generation:

- Variants of Syntax tree
- Three-address code
- Types & Declarations
- Translation of Expressions
- type checking
- Control Flow

Storage Organization

- Stack Allocation of Space
- Access to Non-local data on the Stack
- Heap Management
- Garbage Collection

Intermediate Code Generation



→ In the process of translating a prgm in a given Source lang into code for a given target machine, a Compiler may Construct a sequence of intermediate repr.

Source prgm → High level Intermediate repr → ... → low level Intermediate repr → target code

→ High-level repr are close to the Source lang & low-level repr are close to the target machine.

Variants of Syntax trees

- DAG for Expr
- Value-no method for constructing DAG's

→ Nodes in a Syntax tree represent constructs in the Source pgm

→ the children of a node represents the meaningful components of a construct.

→ The DAG (Directed Acyclic Graph) for an Expr identifies the Common Sub-Expr of the Expr.

DAG for Expression

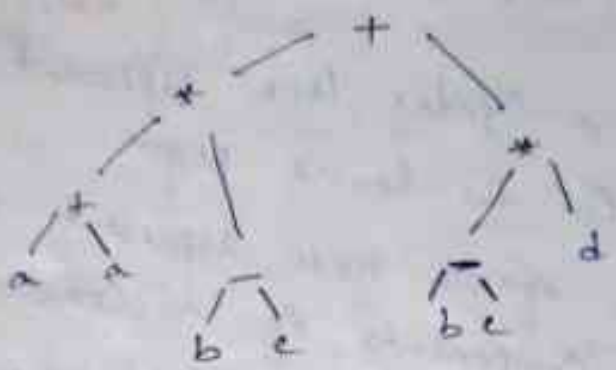
→ like the Syntax Tree for an Expr. A DAG has leaves corresponding to operands & interior nodes corresponding to operators.

→ the difference is that a node N in a DAG has more than one parent if N represents a Common SubExpression.

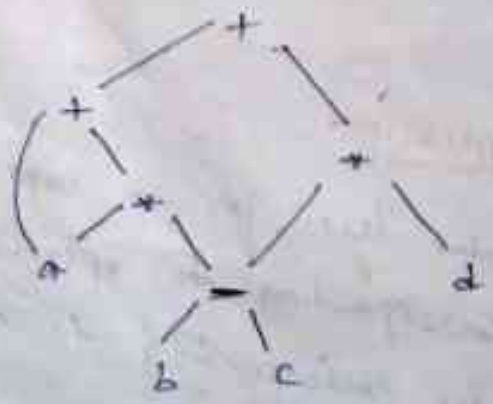
→ in a Syntax tree, the tree for the Common SubExpression would be Replicated as many times as the Sub-Expr appears in the original Expression.

Eg- a + a * (b-c) + (b-c) * d

Draw DAG & Syntax tree



SYNTAX tree



DAG

→ SDD to produce Syntax trees or DAG's

Syntax-Directed Definition

SDD

Production	Semantic Rules
(1) $E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
(2) $E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
(3) $E \rightarrow T$	E.node = T.node
(4) $T \rightarrow (E)$	T.node = E.node
(5) $T \rightarrow id$	T.node = new leaf (id, id.entry)
(6) $T \rightarrow num$	T.node = new leaf (num, num.val)

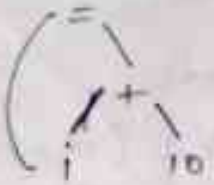
→ steps for Constructing ⟹ syntax tree

1. p_1 = leaf (id, entry-a)
2. p_2 = leaf (id, entry-a) → p_1
3. p_3 = leaf (id, entry-b)
4. p_4 = leaf (id, entry-c)
5. p_5 = Node ('-', p_3, p_4) → {b-c}
6. p_6 = Node ('*', p_1, p_5) → {a * (b-c)}
7. p_7 = Node ('+', p_1, p_6) → {a + a * (b-c)}
8. p_8 = leaf (id, entry-b) → p_3
9. p_9 = leaf (id, entry-c) → p_4
10. p_{10} = Node ('-', p_3, p_4) → p_5
11. p_{11} = leaf (id, entry-d)
12. p_{12} = Node ('*', p_5, p_{11}) → {(b-c) * d}
13. p_{13} = Node ('+', p_2, p_{12}) → {a + a * (b-c) + (b-c) * d}

The Value-no Method for Constructing DAG

→ the nodes of a syntax tree or DAG are stored in an array of records.

Eg:- $i = i + 10$



DAG

1	id		→ Entry for i
2	num	10	
3	+	1	2
4	=	1	3

Array

→ Each row of the array represents one record, & therefore one node.

→ In Each record, the first field is an operation code, indicating the label of the node.

→ In Array, leaves have one additional field, which holds the lexical value (either a Symbol-table pointer or a Constant) and

Interior nodes have two additional fields indicating the left & Right children.

Three - Address Code

Address & Instructions
quadruples ✓
triples
Indirect triples ✓
SSA

→ In three-address code, there is almost one operator on the right side of an instruction.

→ $x + y * z$

$t_1 = y * z$

$t_2 = x + t_1$

Eg:- $a + a * (b-c) + (b-c) * d$

draw DAG & three-address code

DAG

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

Three - address code

Addresses & Instructions

→ Three-address code is built from two concepts:

Addresses

Instructions

→ 3-address code can be implemented using records with fields for the addresses.

Records called
Quadruples
triples &
Indirect - triples

→ An address can be one of the following
- A Name
- A Constant
- A Compiler - generated temporary

→ A Name :-
For convenience, we allow Source-program names to appear as address in three-address code.

→ Constant :- Compiler must deal with many diff types of Constants & Variables.

→ Compiler-generated temporary :-
It is useful, especially in optimizing compiler, to create a distinct name each time a temporary is needed.

COMMON THREE-ADDRESS INSTRUCTIONS FORMS:

1. Assignment Instruction

$x = y$ op z

2. Copy instructions of the form
$x = y$

3. An Unconditional jump goto L

4. Conditional jumps of the form if x goto L

5. Indexed copy inst of the form
$x = y[i]$ &
$x[i] = y$

6. Address & pointer assignments of the form
$x = \&y$,
$x = *y$
$*x = y$

Eg:-
```
do i = i+1 ;
while (a[i] < v);
```
Show types of Labels for 3-addr stmts.

```
L : t1 = i+1
    i = t1
    t2 = i*8
    t3 = a[t2]
    while t3 < v goto L
```

SYMBOLIC LABEL

```
100 : t1 = i+1
101 : i = t1
102 : t2 = i*8
103 : t3 = a[t2]
104 : while t3 < v
      goto 100
```

POSITION NUMBERS or NUMERIC LABEL

QUADRUPLES

→ In a Compiler, Instructions can be implemented as objects or as records with fields for the operator and the operands.

→ A Quadruple (Quad) has four fields, which are op,
arg1,
arg2,
Result

Eg:- $a = \frac{b * -c}{} + b * -c$

Write 3-addr code & Quadruples

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

3-addr code

op	arg1	arg2	Result
minus	c		t_1
*	b	t_1	t_2
minus	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a

Quadruples

Triples

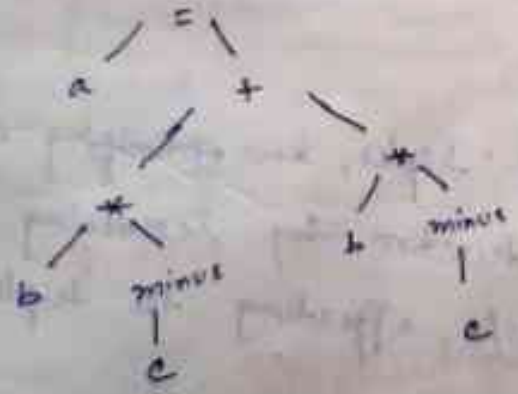
→ A triple has only three fields
op
arg1 ,
arg2

→ Using triples, we refer to the result of an operation x op y by its position, rather than by an explicit temporary name.

→ A triple reps would refer to position (0).

Eg:- a = b * -c + b * -c ;

Draw Syntax tree & Triples



SYNTAX TREE

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Triples

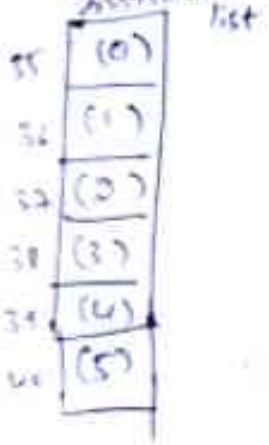
→ Disadvantage

With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.

INDIRECT TRIPLES

→ It consist of a listing of pointers to triples, rather than a listing of triples themselves.

Ex: $a = b * - c + b * - c;$



	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

→ Advantage

With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, w/o affecting the triples themselves.

⇒ STATIC SINGLE-ASSIGNMENT FORM

→ Static Single-Assignment form (SSA) is an intermediate repn that facilitates certain code optimizations.

→ two distinctive aspects distinguish SSA from 3-addr code.

→ the first is that all Assignments in SSA are to variables with distinct names.

→ subscripts distinguish each definition of variables.

((a+b)-c) * d

p = a + b
q = p - c
p = q * d

3-addr code

p1 = a + b
q1 = p1 - c
p2 = q1 * d

SSA form

⇒ Types & Declarations

- type expr
- type equivalence
- declarations
- storage layout for local names

the applications of types can be grouped under checking & translation.

Type checking uses logical rules to reason about the behavior of a program at run time. Especially, it ensures that the types of the operands match the type expected by an operator.

Translation Applications :-

From the type of a name, a compiler can determine the storage that will be needed for that name at run-time.

type info is also needed to calculate the address denoted by an array reference, to insert explicit type conversions.

⇒ Type Expressions

→ types have structure, which we shall represent using Type Expressions :

a type expr is either a basic type or is formed by applying an operator called a Type Constructor to a type expression.

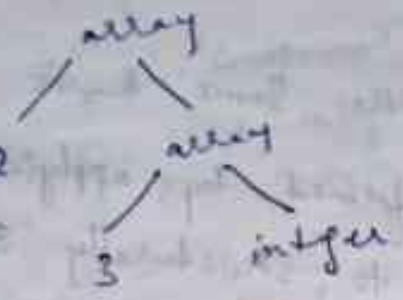
The sets of basic types and constructors depend on the language to be checked.

Eg:- "int [2][3]

read as "array of 2 arrays of 3 integers each"

written as a type expr

array (2, array (3, integer)).



→ A basic type is a type expr.
boolean, char, integer, float & void

→ A type name is a type expression

→ A type expr can be formed by using the type constructor → for function types.

we write $s \to t$ for
"function from type s to type t"

⇒ Type Equivalence

→ when are two type expr equivalent?

→ the key issue is whether a name in a type expr stands for itself or whether it is an abbreviation for another type expr.

→ When type Expr are represented by graphs, two types are structurally equivalent if and only if one of the following condition is true:

1. they are the same basic type.
2. they are formed by applying the same constructor to structurally equivalent types
3. one is a type name that denotes the other

⇒ Declarations

→ types & Declarations using a simplified grammar that declares just one name at a time; declarations with list of names can be handled as.

the grammar is

$D \rightarrow T\ id;\ D \mid \epsilon$

$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$

$B \rightarrow int \mid float$

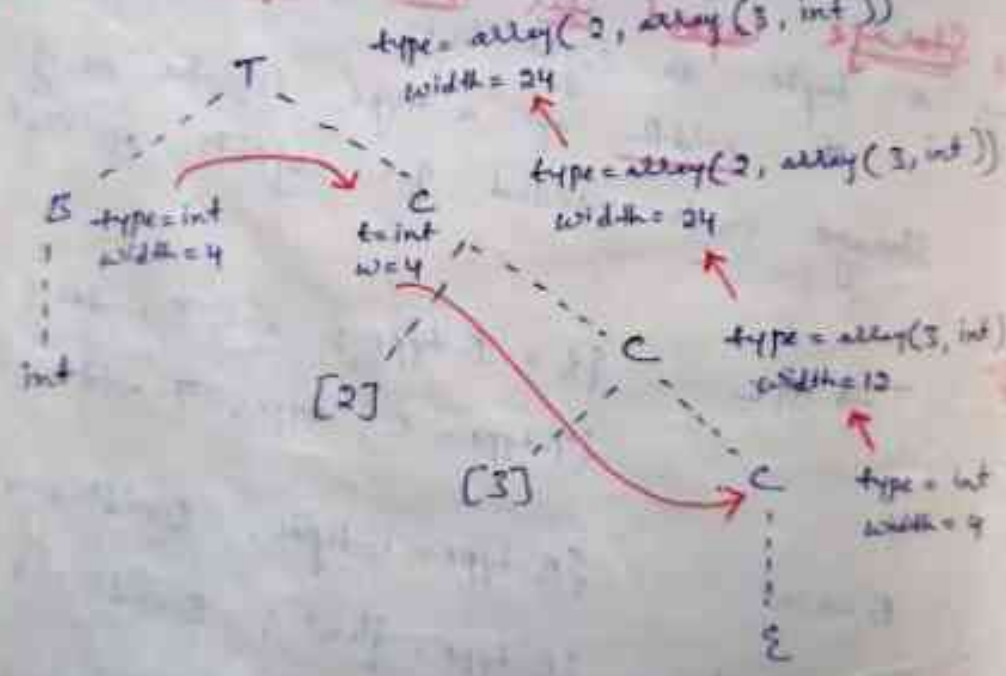
$C \rightarrow \epsilon \mid [num]\ C$

Non-terminal D generates a sequence of declarations.

→ The translation scheme (SDT) computes types & their widths for basic & Array types;

→ SDT uses Synthesized attributes TYPE & WIDTH for each nonterminal & two var t & w to pass type & width info.

parse tree for type int [2][3]



SDT of Array types

Translation of Expressions

operations within Expr
Incremental translation
Addressing Array elements
Translation of Array references

⇒ operations within Expressions

→ three-addr code for an Assignment stmt S using Attribute code for S
attribute addr & code for an

Expr E.

3-addr code for Expr

Production	Semantic Rules
$S \to id = E;$	S.code = E.code ‖ gen(top.get(id.lexeme) '=' E.addr)
$E \to E_1 + E_2$	E.addr = new Temp() E.code = E_1.code ‖ E_2.code ‖ gen(E.addr '=' E_1.addr '+' E_2.addr
$\mid -E_1$	E.addr = new Temp() E.code = E_1.code ‖ gen(E.addr '=' 'minus' E_1.addr)
$\mid (E_1)$	E.addr = E_1.addr E.code = E_1.code
$\mid id$	E.addr = top.get(id.lexeme) E.code = ''

⇒ Incremental Translation

→ the translation scheme generates the same code as the syntax directed def

→ with the incremental approach, the code attribute is not used, since there is a single sequence of inst that is created by successive calls to gen

$S \rightarrow id = E;$ { gen(top.get(id.lexeme) '=' E.addr); }

$E \rightarrow E_1 + E_2$ { E.addr = new Temp(); gen(E.addr '=' E_1.addr '+' E_2.addr); }

| $-E_1$ { E.addr = new Temp(); gen(E.addr '=' 'minus' E_1.addr); }

| (E_1) { E.addr = E_1.addr; }

| id { E.addr = top.get(id.lexeme); }

3-addr code for expr incrementally

⇒ Addressing Array Elements

→ Array elements can be accessed quickly if they are stored in a block of consecutive locations.

→ If the width of each array element is w, then the ith element of array A begins in location

1D

$base + i \times w$

where base = relative address of the storage allocated for the array.

2D

$base + i_1 \times w_1 + i_2 \times w_2$

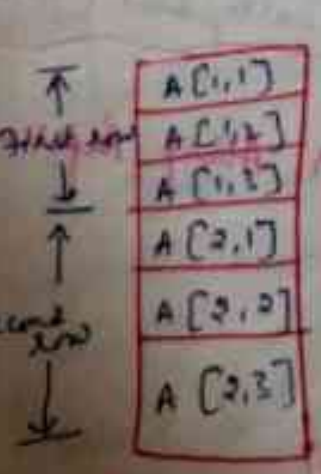
x Dimensions

$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_x \times w_x$

→ A 2D array is normally stored in one of the two forms, either

ROW-MAJOR (row-by-row) or

COLUMN-MAJOR (column-by-column)



ROW MAJOR

2x3 Array

COLUMN MAJOR

⇒ Translation of Array References

→ the chief problem in generating code for array references is to relate the address calculation formulas to a grammar for array references.

→ L generate an array name followed by a sequence of index Expressions.

$L \rightarrow L[E] \mid id \mid [E]$

→ 3-addr code for Expr with array refer

$S \rightarrow id = E;$ gen(top.get(id.lexeme) = E.addr);

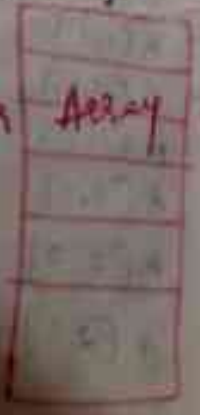
$\mid L = E;$ gen(L.array.base [L.addr] = E.addr);

$E \rightarrow E_1 + E_2$ E.addr = new Temp();
gen(E.addr = E_1.addr + E_2.addr);

$\mid id$ E.addr = top.get(id.lexeme);

$\mid L$ E.addr = new Temp();
gen(E.addr = L.array.base [L.addr];

Semantic Actions for Array References



TYPE CHECKING

Rules for Type checking
Type Conversions
overloading of fns & operators
type inference & polymorphic fns

Type checking

→ to do type checking a compiler needs to assign a type expr to each component of the source pgm.

→ the compiler must then determine that these type exprs conform to a collection of logical rules that is called TYPE SYSTEM for the source language.

→ Type checking has the potential for catching errors in programs.

→ In principle, any check can be done dynamically if the target code carries the type of an element along with the value of the element

→ A SOUND type system eliminates the need for dynamic checking for type errors, because it allows us to determine statistically that these errors cannot occur when the target program runs.

→ An implementation of a language is STRONGLY TYPED if a compiler guarantees that the program it accepts will run w/o type errors.

→ Type checking have been used to improve the security of systems that allow software modules to be imported & executed.

⇒ Rules for Type checking

→ Type checking can take on two forms:
SYNTHESIS &
INFERENCE

→ Type SYNTHESIS builds up the type of an expr from the types of its subexpressions. It requires names to be declared before they are used.

Eg- the type of $E_1 + E_2$
before using this expr, first declare expr E_1, E_2

→ TYPE INFERENCE determines the type of a language construct from the way it is used. type inference is needed for languages

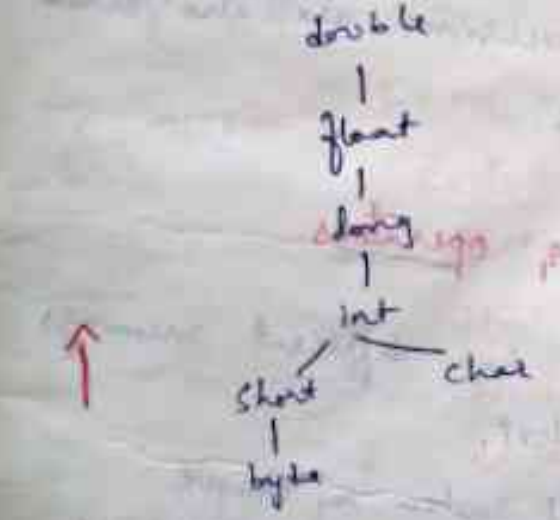
like ML, which check types, but do not require names to be declared.

⇒ Type Conversions

→ Type Conversion rules vary from lang to lang.

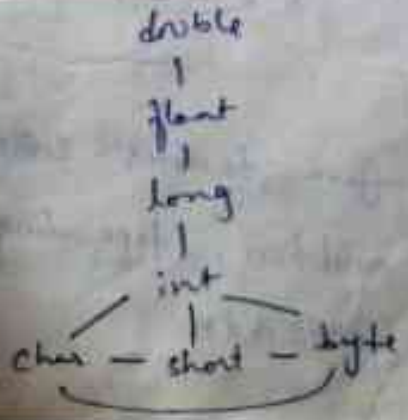
→ In Java, there are two types of conversions
- WIDENING Conversions
- NARROWING Conversions

→ WIDENING Conversions



- Intended to preserve information
- Conversion by Compiler

→ NARROWING Conversions



- which can lose information
- Conversion by programmer

→ Conversion from one type to another is said to be IMPLICIT if it is done automatically by the compiler.

IMPLICIT type conversions, also called as COERCIONS

Coercions, are limited in many languages to widening conversions.

→ Conversion is said to be EXPLICIT if the programmer must write something to cause the conversion.

Explicit conversions are also called CASTS / TYPE CASTING.

⇒ overloading of fns & operators

→ An overloaded symbol has different meaning depending on its context.

→ overloading is resolved when a unique meaning is determined for each occurance of a name.

Eg:- + operator in Java denotes either string concatenation or addition, depending on the types of its operands.

⇒ Type Inference & polymorphic fns

→ Type Inference is useful for a language like ML, which is strongly typed, but doesn't require names to be declared before they are used.

type Inference ensures that names are used consistently.

→ The term polymorphic refers to any code fragment that can be executed with arguments of different types.

CONTROL FLOW ✓

Boolean Expressions
Short-circuit code
Flow of control Stmts
Avoiding Redundant Gotos
Boolean values & jumping code

⇒ Control flow

The translation of stmts such as if-else stmt & while stmt is tied to the translation of boolean expr.

In programming lang, boolean expr are often used to

1. Alter the flow of control :

Boolean Expr are used as conditional expr in stmts that alter the flow of control.

The value of such boolean expr is implicit in a position reached in a prgm.

2. Compute logical values :

A boolean expr can represent T or F as values.

⇒ Boolean Expressions

→ Boolean expr are composed of the boolean operators (&&, ||, !)

(AND, OR, NOT)

applied to elements that are boolean variable or relational expr.

→ Using the attribute Rel.op to indicate which of the six comparison operators <, <=, =, !=, >, >= is represented by Rel.

⇒ Short - Circuit Code

⇒ In SHORT - CIRCUIT (or JUMPING) code, the boolean operators &&, ||, ! translate into jumps.

→ The operators themselves do not appear in the code; instead the value of a boolean expr is represented by a position in the code sequence.

Eg:- The stmt

```
if (x < 100 || x > 200 && x != y)
    x = 0;
```

might be translated into the code

```
    if x < 100 goto L2
    if False x > 200 goto L1
    if False x != y goto L1
L2: x = 0
L1:
```

JUMPING CODE

⇒ FLOW-OF-CONTROL Stmts

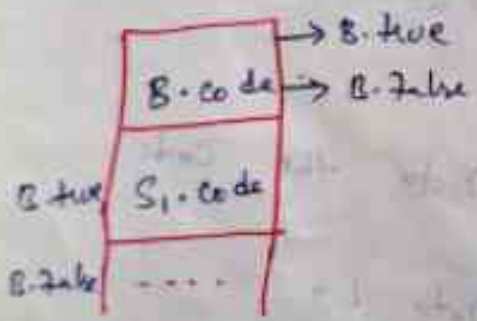
→ The translation of boolean expr into 3-addr code in the context of stmts such as those generated by the following grammar:

$S \rightarrow$ if (B) S_1

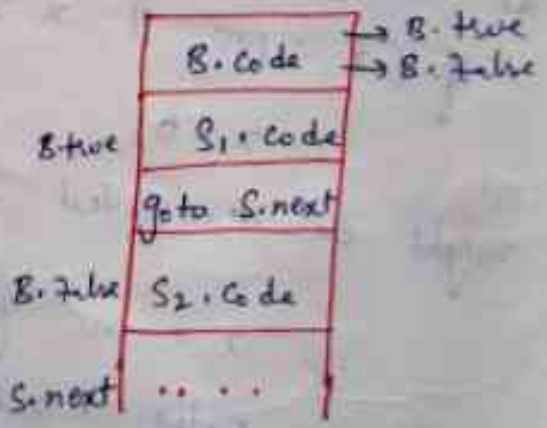
$S \rightarrow$ if (B) S_1 else S_2

$S \rightarrow$ while (B) S_1

The translation



if



if-else

while

⇒ Avoiding Redundant Gotos

if x > 200 goto L1

L1 : goto L4

Instead, consider the rule

if x > 200 goto L4

⇒ Boolean values & Jumping code

1. Use two passes :-
   Construct a complete Syntax tree for the i/p, and then walk the tree in depth-first order, computing the translation specified by the semantic rules.

2. Use one pass for stmt, but two passes for expr :-

RUN-TIME ENVIRONMENTS

Storage Organization / memory org
Stack Allocation of Space
Access to Non-local Data on the Stack
Heap Management
Garbage Collection

1. Memory organiza
2. Static
3. Stack
4. Dynamic (Heap)

1. STORAGE ORGANIZATION

Static Vs Dynamic Storage Allocation

→ the Executing target program runs in its own logical address space in which each program value has a location.

→ The management & organization of this logical address space is shared b/w the compiler, OS & target machine.

→ The OS maps the logical addresses into physical addresses, which are usually spread throughout memory.

→ The run-time representation of an object prgm in the logical address space consists of data

and program area as

⇒ Static Vs Dynamic Storage Allocation

→ Storage-Allocation decision is STATIC, if it can be made by the compiler looking only at the text of the prgm, not at what the prgm does when it executes.

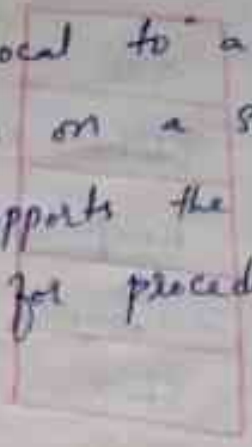
→ A decision is DYNAMIC if it can be decided only while the prgm is running.

→ Many compilers use some combination of the following two strategies for Dynamic Storage Allocation:

* Stack Storage
* Heap Storage

1. STACK Storage

Names local to a procedure are allocated space on a stack.



Stack supports the normal call/return policy for procedures.

2. HEAP storage

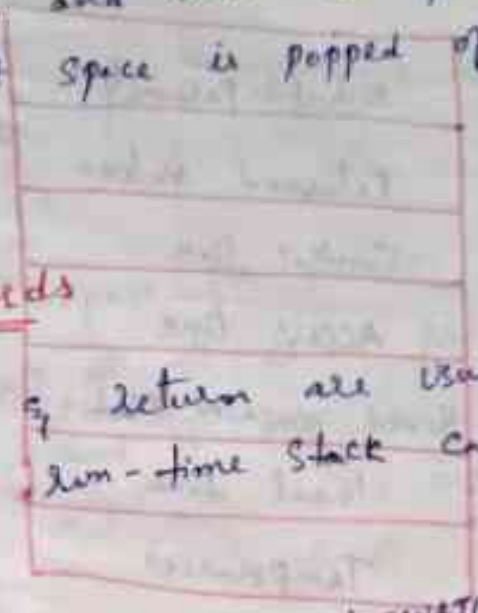
Data that may outlive the ca to the procedure that created it is usually allocated on a Heap of reusable storage.

Heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage whe they are invalidated.

3. STACK ALLOCATION OF SPACE

- Activation Records
- Calling Sequences
- Variable-length Data on 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.



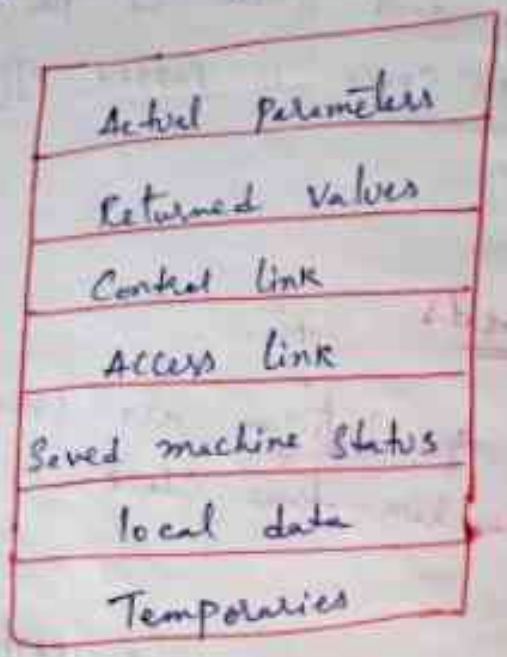
Activation Records

→ procedure calls & returns are usually managed by a run-time stack called the CONTROL STACK.

→ Each live Activation has an ACTIVATION RECORD (frames) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the paths in the activation tree to the activation where control currently resides.

→ the latter activation has its record at the top of the stack.

→ the contents of Activation records vary with the lang being implemented.



Activation Record

1. Temporary values, such as those arising from the evaluation of Expn.
2. Local data belonging to the procedure whose activation record this is.
3. Saved Machine Status, with inf about the state of the machine just before the call to the procedure.
   inf like return addr, contents of registers.

4. Access link may be needed to locate data needed by the called procedure but found elsewhere, in another Activation Record.

5. Control link, pointing to the Activation Record of the caller.

6. Returned values, space for return values if any. Again, not all called procedure return a value, and if one does, we may prefer to place that value in a Register for efficiency.

7. Actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers.

⇒ Calling Sequences

→ procedure calls are implemented by what is known as CALLING SEQUENCES, which consists of code that allocates an Activation Record on the stack and enters info into its fields.

→ Return Sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

Division of tasks b/w caller & callee

⇒ Variable-length Data on the Stack

→ A common strategy for allocating variable-length arrays (i.e. arrays whose size depends on the value of one or more parameters of the called procedure)

ACCESS TO NON-LOCAL DATA ON THE STACK

topic covered in 3rd Unit.

4. HEAP MANAGEMENT / Dynamic Memory

The Memory Manager
Memory Hierarchy of a Computer
locality in programs
Reducing Fragmentation
Manual Deallocation Requests

HEAP Management

→ The heap is the portion of the store that is used for data that lives indefinitely, or until the pgm explicitly deletes it.

→ While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them.

→ For eg, both C++ & java give the programmer new to create objects that may be passed, or pointers to them may be passed, from procedure to procedure, so they continue to exist long after the procedure that created them is gone.

→ Such objects are stored on a heap.

⇒ MEMORY MANAGER

→ The memory manager keeps track of all the free space in heap storage at all times.

→ It performs two basic jobs:

Allocation

Deallocation

- Allocation :- When a program requests memory for a variable or object, the memory manager produces a CHUNK of contiguous heap memory of the requested size.

- Deallocation :- The Memory manager returns deallocated space to the pool of free space, So it can reuse the space to satisfy other allocation requests.

PROPERTIES of Memory Managers :-

- 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.

⇒ Memory Hierarchy of a Computer

>2GB	Virtual Memory	3-15ms
256MB-2GB	physical Memory	100-150ns
128KB-4MB	2nd level cache	40-60ns
16-64KB	1st level cache	5-10ns
32 words	Registers (processor)	1ns

⇒ Locality In Programs

→ Most programs exhibit a high degree of LOCALITY ; that is, they spend most of their time executing a relatively small fraction of the code & touching only a small fraction of the data.

→ Program has two types of locality
TEMPORAL locality
SPATIAL locality

→ TEMPORAL locality :-
we say that a program has temporal locality if the memory location it accesses are likely to be accessed again within a short period of time.

→ SPATIAL locality :-
we say that a program has spatial locality if memory locations close to the location accessed are likely also to be accessed within a short period of time.

→ Pgms spend 90% of their time executing 10% of the code.

→ Reducing Fragmentation

→ At the beginning of pgm execution, the heap is one contiguous unit of free space.

→ As the pgm allocates & deallocates memory, this space is broken up into free & used

chunks of memory, and the free chunks need not reside in a contiguous area of the heap.

→ we refer to the free chunks of memory as HOLES.

→ With each allocation request, the memory manager must PLACE the requested chunk of memory into a large-enough hole.

→ Unless a hole of exactly the right size is found, we need to split some hole, creating a yet smaller hole.

→ With each deallocation request, the freed chunks of memory are added back to the pool of free space.

→ we COALESCE contiguous holes into larger holes, as the holes can only get smaller otherwise.

→ If we are not careful, the free memory may end up getting fragmented, consisting of large no. of small, noncontiguous holes.

→ It is then possible that no hole is large enough to satisfy a future request, even though there may be sufficient aggregate free space.

Best-fit & Next-fit object placement

Best-fit algorithm tends to spare the large holes to satisfy subsequent, larger requests.

Managing & coalescing free space

There are two data structure that are useful to support coalescing of adjacent free blocks:
- Boundary Tags
- Doubly linked, Embedded Free list

- **Boundary Tags:**

At both ends, we keep a free/used bit that tells whether or not the block is currently allocated (used) or available (free).

Adjacent to each free/used bit is a count of the total no. of bytes in the chunk.

- **Doubly linked, Embedded Free list:-**

The free chunks are also linked in a doubly linked list.

The pointers for this list are within the blocks themselves, say adjacent to the boundary tags at either end.

Thus, no additional space is needed for the free list, although its existence does place a lower bound on how small chunks can get; they must accomodate two boundary tags & two pointers, even if the object is a single byte.

⇒ Manual Deallocation Requests

Problems with Manual Deallocation :-

manual memory mgt is error-prone.

The common mistake take two forms:

Failing ever to delete data that cannot be referenced is called a MEMORY LEAK error.

~~Referring~~ Referencing deleted data is a DANGLING-POINTER-DEFERENCE error.

Programming Conventions & Tools

Tools are

Object Ownership

Reference counting

Region-based Allocation

OBJECT OWNERSHIP

it is useful when an object's lifetime can be statically reasoned about

The idea is to associate an OWNER with each object at all times.

The owner is a pointer to that object, presumably belonging to some fn invocation.

The owner is responsible for either deleting the object or for passing the object to another owner.

However, it doesn't help solve the dangling-pointer-reference problem, because it is possible to follow a non-owning pointer to an object that has been deleted.

Reference Counting

it is useful when an object's lifetime needs to be determined dynamically

The idea is to associate a count with each dynamically allocated object.

Whenever a reference to the object is created, we increment the reference count

whenever a Reference to the object is created, we Increment the Reference Count; whenever a Reference is removed, we decrement the Reference Count.

When the Count goes to zero, the object can no longer be referenced and can therefore be deleted.

Region- Based Allocation

it is useful for collections of objects whose lifetimes are tied to specific phases in a Computation.

When objects are created to be used only within some steps of a Computation, we can allocate all such objects in the same region.

Region-based allocation technique has limited applicability.

GARBAGE COLLECTION

Design Goals for Garbage Collectors
Reachability
Reference Counting Garbage Collectors

→ Data that cannot be referenced is generally known as GARBAGE.

→ Many high-level prgming languages remove the burden of manual memory mgt from the prgrmmer by offering automatic garbage collection, which deallocates unreachable data.

⇒ Design Goals

→ Garbage Collection is the reclamation of chunks of storage holding objects that can no longer be accessed by a prgm.

→ we need to assume that objects have a type that can be determined by the garbage collector at run time.

→ From the type information, we can tell how large the object is and which components of the object contain references to other objects.

→ A user prgm, which we shall refer to as the MUTATOR, modifies the collection of objects in the heap.

→ The mutator creates objects by acquiring space from the memory manager, and the mutator may introduce and drop references to existing objects.

→ objects become garbage when the mutator prgm cannot reach them.

type Safety
performance Metrics - Reachability

⇒ Type Safety

→ A language in which the type of any data component can be determined is said to be TYPE SAFE.

→ There are type-safe languages like ML, for which we can determine types at compile time.

→ Java, type will be determined at run-time. Called DYNAMICALLY typed language.

→ lang which is neither statically nor dynamically type safe, then it is said to be UNSAFE.

Performance Metrics

performance metrics that must be considered when designing a garbage collector are:

- overall Execution time
- Space Usage
- pause time
- prgm locality

Reachability

→ we refer to all the data that can be accessed directly by a prgm, w/o having to dereference any pointer, as the ROOT set.

→ Reachability becomes a bit more complex when the prgm has been optimized by the Compiler.

There are four basic operations that a Mutator performs to change the set of reachable objects:

- object Allocations
- parameter passing & Return values
- Reference Assignments
- procedure Returns

→ Reference Counting Garbage Collectors

→ Garbage Collector identifies garbage as an object changes from being reachable to unreachable;

→ the object can be deleted when its count drop to zero.

→ With a reference-counting garbage collector, every object must have a field for the reference count.

→ Reference counts can be maintained as follows:

Object Allocation:- the reference count of the new object is set to 1.

parameter passing:- the reference count of each object passed into a procedure is incremented.

Reference Assignments:- for stmt U=V, where U & V are references, the reference count of the object referred to by 'V' goes up by one, and the count for the old object referred to by 'U' goes down by one.

Procedure Returns :- As a procedure exits, objects referred to by the local variables in its activation record have their counts decremented.

Transitive loss of Reachibility :- whenever the reference count of an object becomes zero, we must also decrement the count of each object pointed to by a reference within the object.

PDF Created Using

Camera Scanner

Easily Scan documents & Generate PDF

https://play.google.com/store/apps/details?id=photo.pdf.maker