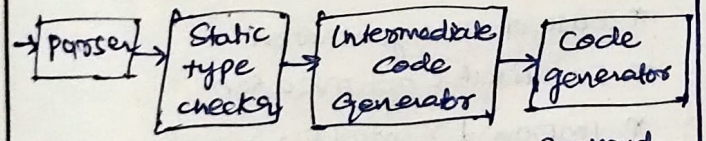
KESHAV MEMORIAL ENGINEERING COLLEGE

Expt. No._____
Date_____

CD4:
Intermediate code Generation: Variants of syntax trees, three-Address code, Types and Declarations, Translation of Expressions, Type checking, Control Flow, Backpatching, Switch statements, Intermediate code for procedures.
Run-time environment: Storage organization, Stack Allocation of space, Access to Non local Data on the stack, parameter passing, Heap management and Garbage collection.

Intermediate Code Generation:

* converting source program into machine code in one pass is not possible always. So compiler generates an easy representation of source language (program) called as Intermediate language which leads to an efficient code generation.



Forms of Intermediate Code

1. Abstract Syntax Tree
2. Polish Notation.
3. Three Address Code.

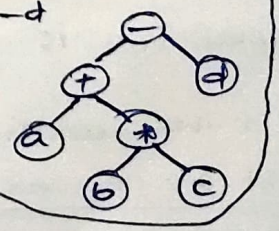
Variants of Syntax trees:

Directed Acyclic Graphs (DAG's)

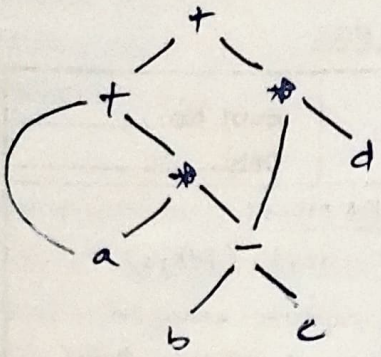
* It efficiently capture common subexpressions and optimize their evaluation.
* Initially, expression in the source code are represented as [ASTs]
  ↓
  these can contain redundant nodes representing common subexpressions.
* The compiler traverses the expression trees and constructs a DAG by identifying common subexpressions and sharing their nodes.
* Each node in the DAG represents an operation / a variable.
* Edges represent dependencies between the nodes.
* Optimizations used: Constant folding, algebraic simplification and common subexpression elimination.
* These optimizations help reduce the size of the DAG and improve the efficiency.
* The DAG serves as an intermediate representation that simplifies subsequent stages of compilation process while preserving the semantics of the original program.

It is condensed form of parse trees. Ex: a+b*c-d

(leaf → operand
node → operator.



Ex:- a+a*(b-c)+(b-c)*d



Polish Notation:

* It is a linearization of AST.
* Most natural way
* also called prefix expression.

(operator op1 op2)

Ex:

(a+b)*(c-d) ⇒ *(+ab)(-cd)

⇒ *+ab-cd.

Reverse Polish Notation (postfix evaluation)

Ex:

(a+b)*(c-d) ⇒ (ab+)(cd-)*

⇒ ab+cd-*

Three-Address Code:

* built from two concepts called addresses and instructions.
* given expression is broken down into several seperate instructions.
* these instructions can easily translate into assembly language.

General form is a:=b op c

* In a three-address code there is at most one operator at the right side of an operation.

Ex:-

a+a*(b-c)+(b-c)*d t1=b-c

a+a*t1+t1*d t2=a*t1

a+t2+t1*d t3=a+t2

t3+t1*d t4=t1*d

t3+t4 t5=t3+t4

t5.

Three-Address code:

* It is a sequence of statements of the general form

x := y op z

→ names,
→ constants,
→ compiler-generated temporaries

→ any operator
→ fixed / floating point arithmetic operator
→ logical operator

Ex: x+y*z

t1 := y*z

t2 = x+t1

↳ temporary names.

Advantages:-

* Simplicity
* Platform Independence
* Ease of code generation
* Facilitates optimization
* Improved Control Flow Analysis
* Three Address statements can be implemented as records with fields for the operator and the operands.

3-Such representations are:

* Quadruples
* Triples
* Indirect triples.

Expt. No.
Date

Quadruples :

* record structure with four fields which are op, arg1, arg2 and result.

Ex: $X := y \ op \ z$

$X := y * z$

$(*, y, z, X)$:

↳ entered in symbol table

Triples

* refer to a temporary value by the position of the statement that computes it.
* record structure with three fields op, arg1 and arg2.

Ex: $X := y * z$

$(*, y, z)$:

Indirect Triples :

* listing pointers to triples, rather than listing the triples themselves.

Ex: $b * -c + b * -c$

Three-address code.

$t1 = \text{minus } c$

$t2 = b * t1$

$t3 = \text{minus } c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

* 3-address code simplifies the task of generating machine code.

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

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

	Quadruples			
	op	arg1	arg2	result
	minus	c		t1
	*	b	t1	t2
	minus	c		t3
	*	b	t3	t4
	+	t2	t4	t5
	=	t5		a

Types and Declarations:

* Data type of a name / identifier and type checking of the expression / Statement is to be done before Intermediate code Generation.
* type checking uses logical rules to reason about the behaviour of a program at run time.
  - ⇒ it ensures that the types of the operands match the type expected by an operator.
* Type information is also needed
  - ⇒ to calculate the address denoted by an array reference.
  - ⇒ To insert explicit type conversions
  - ⇒ To choose the right version of an arithematic operator.

Basic Types: integer, character, real, Boolean.

Derived types: Array, Structure (record), set and pointer.

Declarations:

* declares just one name at a time.

Generates

D → T id ; D | ε → Sequence of declarations

T → B C | record '{' D '}' → basic, array or record types.

B → int | float → basic int, float types.

C → ε | [num] C → strings of zero or more integers.

Type expressions

* Types have Structure, represented by using type expressions.
* A type expression is either a basic type or it is formed by applying an operator called a type constructor to a type expression.
* Type constructors are derived types (array, structure, pointer function).

Type Expression examples:

Array: int array[20]

⇒ array(1,2 ... 19, int)

int[2][3] → array(2, array(3, integer))

Product: t1 * t2 (* is left associative)

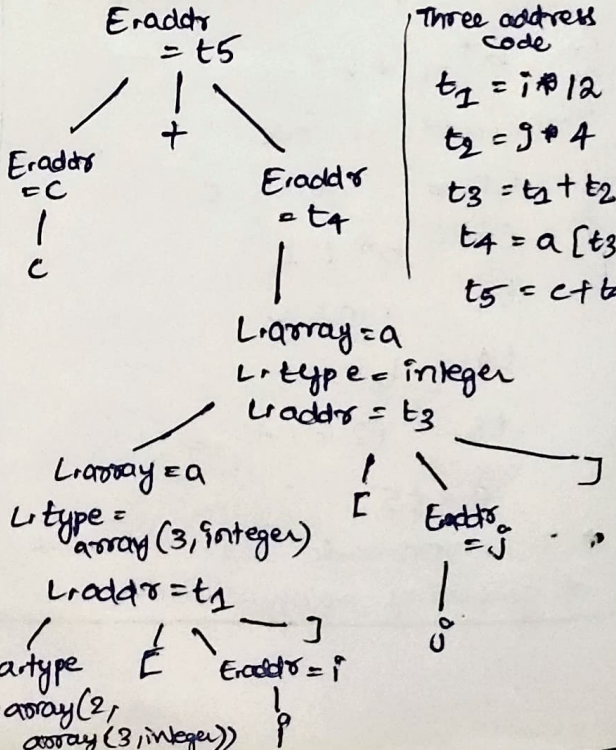
pointer: float *y; pointer(float).

Type equivalence:

* if two type expressions are equal then return a certain type else error.
* When type expressions are represented by graphs, two types are Structurally equivalent if and only if one of the following is true:
  - ⇒ If they are of the same basic type.
  - ⇒ If they are formed by applying the same constructor to Structurally equivalent types.
  - ⇒ one is a type name that denotes the other.

Translation of Array References:

Annotated parse tree for c + a[i][j]



Expt. No.
Date

CONTROL FLOW

* The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions.

Boolean expressions are often used to:

⇒ compute logical values.
⇒ Alter the flow of control.
⇒ Short circuit evaluation.
⇓ Boolean operators (&&, ||, !)
⇓ Relational expressions are of form
→ E1 rel E2

(arithmatic exp) (<, <=, =, !=, >, >= is represented by rel)

Flow of control statements:

* In the below grammar, non-terminal B represents a boolean expression and non terminal S represents a statement.

S → if (B) S1
| if (B) S1 else S2
| while (B) S1

B.true = newlabel()
B.false = S.next()
S1.next = S.next

S.code = B.code || label (B.true) || S1.code,

Type checking

* A compiler needs to assign a type expression to each component of the source program.
* Type checking can take care of the two forms:
⇒ Synthesis ⇒ Inference.

Type synthesis: builds the type of ~~an~~ expression from the types of its subexpressions.

* A typical rule for type synthesis has the form
"if f has type S → t and x has type S, then expression f(x) has type t."

f and x denote expressions.
S → t denotes a function.

* The rule can be adopted for E1+E2 by viewing it as a function application
add (E1, E2). ⇒ E.type

Type inference: determines the types of a language construct from the way it is used.

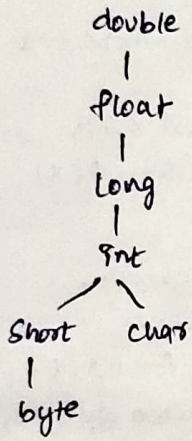
* some statements do not have values. so, void type is used.
* A typical rule for type inference has the form.
"if f(x) is an expression, then for some α and β, f has type α → β and x has type α"
* rule:
S → if (E) S1 :
if E.type = boolean true,
S.type = S1.type

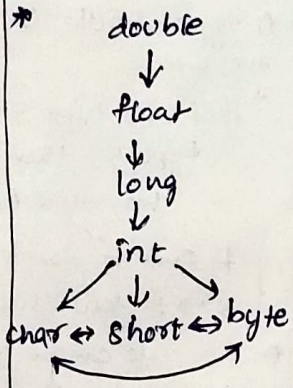
Type conversion

```
2 * 3.14
t1 = (float)2      // explicit.
t2 = t1 * 3.14
```

Two types of conversions happen:

Widening Conversion.	Narrowing Conversion.
* intended to preserve information.	* which can lose information.
* any type lower in the hierarchy can be widened to a higher type.	* a type 's' can be narrowed to a type t if there is a path from s to t.





Implicit → coercions
Explicit → casts

Backpatching

* Generation of three address code in a single pass creates a problem in identifying addresses of the label statements.

Ex:- Jump statements (goto (Boolean))
flow of control statements.

Def:- Backpatching is the activity of filling up unspecified / missing information of labels using appropriate semantic actions during the code generation phase.

* In Boolean expressions we must insert symbolic labels for jumps.
  ⇒ So, Boolean expressions needs a seperate pass to set them to appropriate addresses for labels.
* Backpatching is needed to avoid the need of two passes.
  ⇒ Here we save the sequence of instructions into an array and labels will be indices of the array.
* For Non-Terminal, we use two attributes
  ⇒ truelist : it contains the addresses of the true instructions.
  ⇒ falselist : it contains the addresses of the false instructions.
* A new non-terminal called M (marker)
  M → ∈, which gives the address of missing / unspecified label.
* To generate code using backpatching:
  → makelist(i)
  → merge(p1, p2)
  → backpatch(p, i)

Switch statements

```
switch (E)
{
    case 1 : S1
             break;
    case 2 : S2
             break;
    default : S3
}
```

three-address code for switch statement:

```
evaluate E value
    go to test
L1 : S1
     goto last
L2 : S2
     go to last
L3 : S3
     go to last
test : if (E==1) goto L1
```

Expt. No.
Date

```
if (E==2) goto L2
goto L3  // default
Last: end.
```

Intermediate code for procedures.

In three-address code, a function call is specified with the below steps.

→ Evaluation of parameters in preparation for a call (pass by value).

→ Function call itself.

Ex:1 `n = f(a[i]);`

(f → function, a → array)

might translate into

```
1) t1 = i * 4
2) t2 = a[t1]
3) param t2
4) t3 = call f, 1   // 1 indicates no. of args of function f.
5) n = t3.
```

Ex:2

```
int f(int x, int y)
{
    return x+y+1;
}
```

Function call: f(2+3, 4)

```
1) t1 = x
2) t2 = y
3) param t1
4) param t2
5) t3 = call f, 2
```

Run-Time Environment:

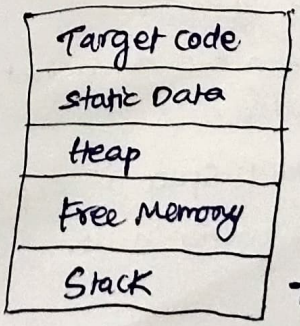
* A compiler must accurately implement the abstractions like names, scope, bindings, data types, operators, procedures, and flow-control statements of source language.

The environment deals with:

* Layout and storage allocations for objects.
* Mechanism of accessing variables.
* Linkage between procedures.
* parameter passing mechanism.
* Memory management (stack & heap)
* Garbage collection.

Storage Organization:

* Compiler demands a block of memory from operating system to run the compiled program. This block is called as run time storage.
* Run-time storage comes in blocks of contiguous bytes divided into parts to hold code and data.
  - → Generated Target code.
  - → Data objects.
  - → Information which keeps track of procedure activations.



* Size of the generated target code is fixed (so, placed in static space).
* Memory required for data objects is known at compile time.
* Stack is used to manage active procedures.
* Heap stores other information.
* Size of stack and heap is not fixed.

Storage Allocation Strategies:

1. Static → for all data objects at compile time.
2. Stack → used to manage the run time storage.
3. Heap → used to manage the dynamic memory allocation.

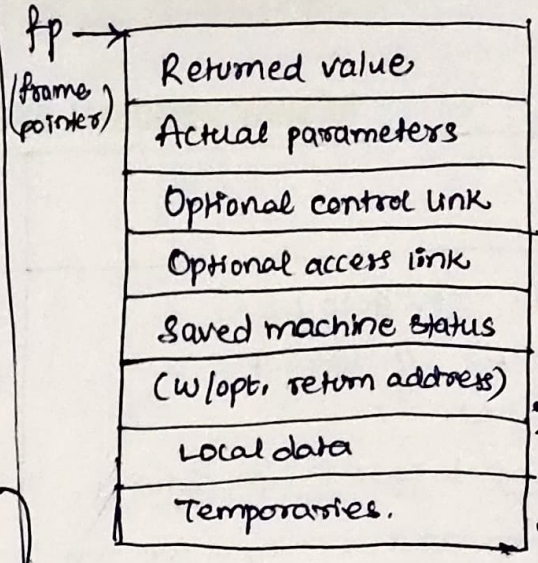
Pros: easy for the compiler to find the address of the objects in activation record.

Cons: ⇒ Knowing the size of data object at compile time is mandatory.
⇒ Data structures cannot be created dynamically.
⇒ Recursive procedures are not supported.

* Also called control stack.
* Each time a function is called, space for its local variables is pushed onto a stack.
* Local variables are stored in activation record and bound to it.
* Data structures can be created dynamically.

Stack Allocation of Space:

* For every function call an activation record is created.
* Allocation of stack space is depends on number of function calls.



Activation record: Also known as a stack frame / activation frame, is a data structure used to manage the execution of a subroutine or function within a program.

⇒ It typically contains various fields that store information such as:

Returned value: When the subroutine or function finishes, the value stored in this field is passed back to the caller.

Actual parameters: When a subroutine or function is called, the actual parameters passed to it are stored in the activation record.

Optional control link: This link points to the activation record of the caller subroutine or function.

Optional access link: It points to the activation record of the nearest enclosing scope or outer subroutine.

Saved machine status (with optional return address): It typically consists of the values of CPU registers that need to be preserved across subroutine calls.

Expt. No.
Date

Local data : Space is allocated within the activation record to store local variables declared within the subroutine / function.

Temporaries : are typically used for intermediate calculations & storage within the subroutine & function.

Limitation :

* Stack allocation would not be feasible if function calls did not nest in time.
* Memory addressing is slow because of pointers and index registers.

Heap Allocation

* If nonlocal variables must be retained even after the activation record, then heap allocation is required.
* Heap allocations allocates continuous blocks of memory when required for storage of activation records / other data obj.
* It can be deallocated when activation ends, and it can be reused by heap manager.
* Linked list is suitable for efficient Heap Management.

Access to Nonlocal Data on the Stack.

⇒ Storage allocation is done for two types of data : Local and Non-Local data.

⇒ If a procedure refers to variables that are not local to it, such variables are called non-local variables.

⇒ Global variables are placed in static storage, storage and location is known at compile time.

⇒ Accessing these variables is easy, Static scope is used to access non-local names.

⇒ In some languages, which support nesting of procedures. In these languages non-local data are handled by using scope information.

⇒ Two types of scope rules :

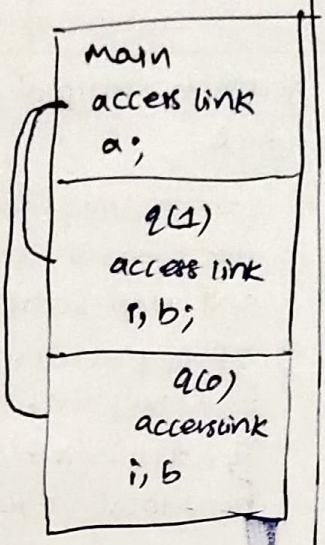
* Static scope (used by block structured language).
* Dynamic scope (used by non-block structured language).

Static Scope / Lexical Scope :

* Scope is verified by examining the text of the program.
* If procedures are nested, nesting depth of the procedure is used.

* The lexical scope of nested procedures can be implemented by using Access links and Display.

Display :- Faster to access non-local names.



Dynamic Scope:

* used in non-block structured languages (LISP).
* By considering the current activation, it determines the scope of declaration of the names at runtime.
* In this type, the scope is verified at runtime.

Deep Access: The idea is to keep a stack of active variables.

* use control links instead of access links.
* to find a variable, search the stack from top to bottom.

Shallow Access :- The idea to keep a central storage and allot one slot for every variable name.

Parameter passing:

* If one procedure calls another, communication between them is made by using non-local names and parameters.

Actual params : variables specified in function calls.

Formal params : variables declared in function definition.

L-value → refers to storage.

R-value → refers to value contained in the storage.

Parameter passing Techniques:

1. call by value.
2. call by reference. } +
3. copy restore ← ↙
4. call by name
   (Actual params are substituted for the formals).

Heap management:

* The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.
* Local variables typically become inaccessible when their procedures end.
* Memory Manager is a subsystem that allocates and deallocates space within the heap.
  1) Interface b/w app prog & OS.
  2) always keep track of the free space in heap storage.
  3) It performs:
     1. Allocation
     2. Deallocation.
* Memory Management would be simpler if
  (a) all allocation requests were for chunks of the same size, and
  (b) storage was released predictably, say, first-allocated first deallocated.
* Desirable properties of memory manager:
  1) Space efficiency.
  2) Program u.
  3) Low overhead.

Expt. No
Date

Memory hierarchy in runtime environment: