MACROS

Macro definition

Macro Allows a sequence of source language code to be defined once and then referred to by name. Each time it is to be referred. Each time this name occures in a program, the sequence of codes is substituted at that point

Macros are defined at the start of program
A macro definition consist of
1. MACRO pseudo opcode
2. Name of macro
3. List of parameteres.

4. body of Macro (instruction) or sequence to be abbreviated

5. MEND pseudo opcode.

MACRO instruction

MACRO — start of definition
[ ] — name of Macro
{ - - - — sequence to be abbreviated
MEND — End of definition

eg:

```
MACRO
INCR
ADD 1, DATA
ADD 2, DATA
ADD 3, DATA
MEND
INCR
INCR
```

Macro definition.

Macro call

Features of Macro facilities

(How many ways we can write macros)

1) Macros with arguments

Page No. :
Date / /

2. Conditional macro expansion
3. Macro call within macro definition
4. Macro within macro definition (Nested macro)

1) Macros with arguments

Source code

```
MACRO
&LAB INCR &ARG0, &ARG1, &ARG2
&LAB ADD AREG, &AREG1
     ADD AREG, &ARG2
     ADD AREG, &ARG3
MEND
START
LOOP INCR A, B, C // macro call
LABEL INCR DATA, DATA2, DATA3
A DC2
B DC 2
CDS 2
DATA1 DC 3
DATA 2 DC 2
DATA 3 DC 4
END
```

Definition

Expanded code

```
START
LOOP ADD AREG, A
     ADD AREG, B
     ADD AREG, C
LABEL ADD AREG, DATA
      ADD AREG, DATA2
      ADD AREG, DATA3
```

Date / /

Macro and different types of parameters

positional parameter

```
MACRO
&LAB  INCR  &ARG0, &ARG1=X, &ARG2=Y
&LAB  ADD   AREG, &ARG1
      ADD   AREG, &ARG2
      ADD   AREG, &ARG3
      MEND
```

Keyword parameter

1. Positional parameter: Is written as & parameter name.
   eg: &LAB and &ARG0 are positional parameter.
2. Keyword parameter: are used for assigning default values to the parameter.
   eg: &ARG1 with default value X, &ARG2 with default value Y.
3. Actual parameter :- when macro call is defined each actual parameter will be used

```
      LOOP  INCR  DATA
LOOP  INCR  DATA1, DATA2, DATA2
```

```
MACRO
&LAB  INCR  &ARG0, &ARG1=X &ARG2=Y
&LAB  ADD   AREG, &ARG1
      ADD   AREG, &ARG2
```

ADD AREG1, &ARG3
MEND

II conditional Macro expansion

Source code

```
MACRO
&ARG0 VARY &count, &ARG1, &ARG2, &ARG3
&ARG0 ADD AREG1, &ARG1
AIF(&COUNT EQ1). FIN1
ADD BREG1, &ARG2
AIF (&COUNT EQ2). FIN1
ADD CREG1, &ARG3
FIN1 MEND
    START
L1  VARY 3, X, Y, Z
L2  VARY 2, MIN, DATA
L3  VARY 1, DATA
```

Extended form

```
    START
L1  ADD AREG1, X
    ADD BREG1, Y
    ADD CREG1, Z
L2  ADD AREG1, M
    ADD BREG1, N
L3  ADD AREG1, DATA
```

3. MACRO call within macro definition Expanded code

Source code

```
MACRO
ADD1 &ARG1
LOAD AREG1, &ARG1
ADD AREG1 = F"1"
STORE AREG1, &AREG1
MEND

MACRO
ADDS &ARG1, &ARG2, &ARG3
    ADD1 &ARG1
    ADD1 &ARG2
    ADD1 &ARG3
    MEND
START
ADDS X,Y,Z

XDC2
YDC2
ZDS2
LDC2
MDS2
NDC4
END
```

Expanded code

```
START
LOAD AREG1, X
ADD AREG1 = F"1"
STORE AREG1, X

LOAD AREG1, Y
ADD AREG1, = F"1"
STORE AREG1, Y

LOAD AREG1, Z
ADD AREG1, = F"1"
STORE AREG1, Z

XDC2
YDC2
ZDS2
LDC2
MDS2
NDC4
END
```

4 Macro definition within macro definition

```
MACRO
DEFINE &SUB
    MACRO
    &SUB &Y
    CNOP 0, 4
    BAL 1, *+8
    DC A(&Y)
    L 15, =V(&SUB)
    BALR 14, 15
    MEND
    MOVER AREA, X
    MEND
```

Lexical Expansion/substitution – 1. Replace call by model
2. Replace formal parameter by actual parameter

1) Positional parameter expansion
2) Keyword parameter expansion
3) Default parameter expansion

Page No. :
Date / /

2) Positional parameter expansion

eg:

```
Macro pgm
START 300
. . . .
. . . .
. . . .
INCR A,B,CREG1
. . . . .
. . . .
. . . .
END
```

Definition of MACRO INCR

```
MACRO
INCR &MEM, &VAL, &R        (formal)
MOVER &R, &MEM             (Model stmt)
ADD &R, &VAL
MOVEM &R, &MEM
MEND
```

↓

```
MOVER CREG1,A
ADD : CREG1,B
MOVEM CREG1 A
```

↑ expansion of macro INCR inside main program

Formal Parameter	Actual Parameter
MEM	A
VAL	B
R	CREG1

Page No. :
Date / /

2) Keyword parameter Expansion

eg :

```
START 300
    .
    .
INCR VAL=Y R=AREG1, MEM= X
    .
    .
END
```

formal parameter	Actual parameter
MEM	X
VAL	Y
R	AREG1

Definition of MACRO INCR

```
MACRO
INCR &MEM=, &VAL=, &R=
MOVER &R, &MEM
ADD &R, &VAL
MOVEM &R, &MEM
MEND
```

Expansion of macro INCR inside main pgm

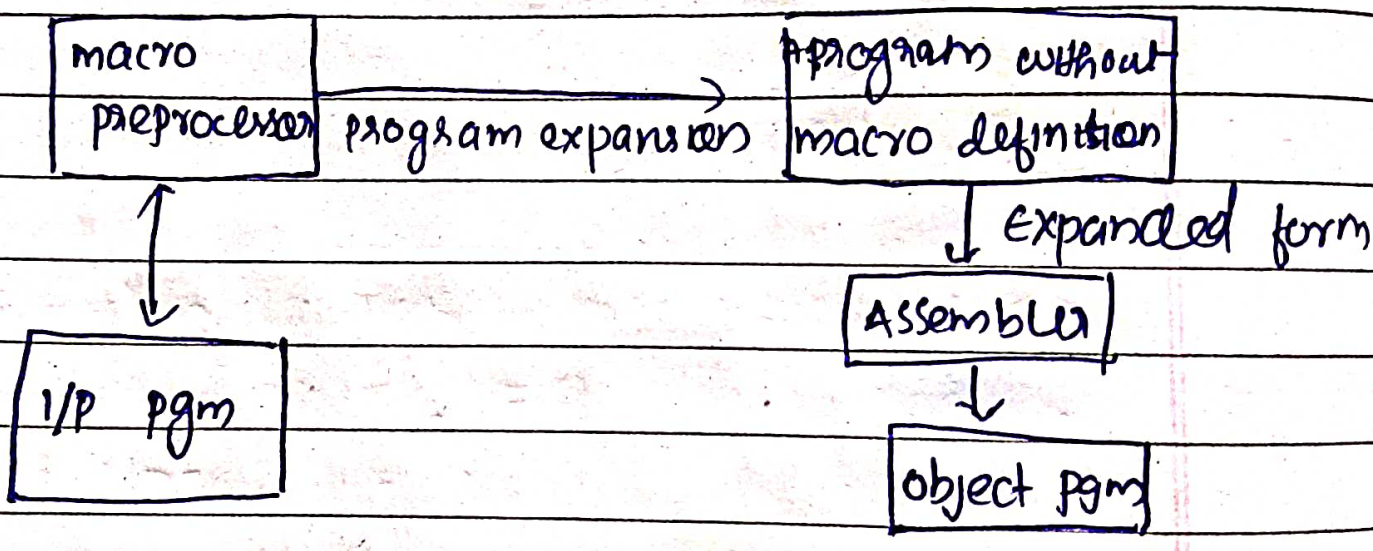
```
MOVER, AREG1, X
ADD AREG1, Y
MOVEM AREG1, X
```

Default parameter expansion

* in definition define default values
* Its useful when most of call contain same values
* If still parameter passed, it will overwrite default value.

MACRO PRE PROCESSOR

MACROpreprocessors are vital for processing all programs contain macro definitions or calls. A macropreprocessor essentially accepts assembly program with macro definitions and calls as its input & process it into an equivalent expanded assembly program with no more definitions and calls. The macro preprocessor output pgm is then passed ones to an assembler to generate target object pgm. The general design semantics of a macro preprocessor is



The following are involved in macro expansion

i) recognising a macro call
ii) deciding what values formal parameter should have in the expansion of macro call
iii) Maintaining values of expansion time variable

during the expansion of macro call.
* organising expansion time control flow
* finding the stmt that defines a specific sequencing symbol
performing expansion of a model statement.

Datastructures of Microprepocessor

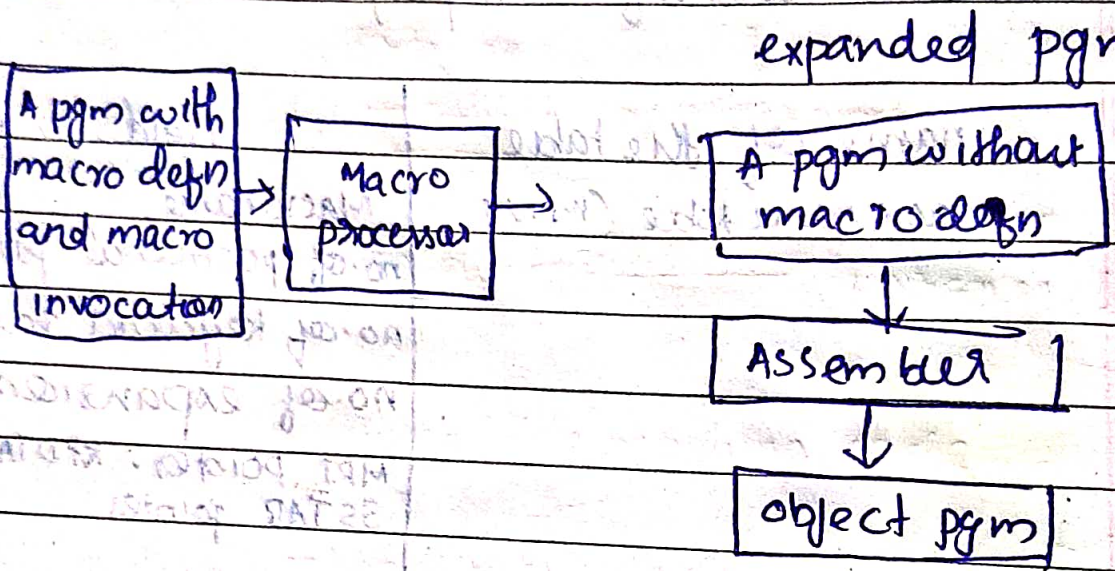
	name of the table	Fields in each entry
i)	Macro name table (MNT	Macroname, no. of positional parameter no. of keyword parameter no. of expansion time variable MDT pointer. KPDTAB pointer SSTAB pointer
ii)	parameter name table (PNTAB)	Parameter name
iii)	Expansion time variable name table (ENVTAB)	Expansion time variable name.
iv)	Sequencing symbol name table (SSNTAB)	Sequencing symbol name
v)	keyword parameter default table	parameter name, default value
vi)	Macro definition table (MDT)	label, opcode, operand
vii)	Actual parameter table (APTAB)	value
viii)	expansion time variable table	value
ix	sequencing symbol table	MDT entry

Page No. :
Date / /

Two pass macroprocessor :

Explain design of 2 pass MACRO processor in detail.

Macropreprocessor takes a source program containing macro definitions and macro calls. It translate it into assembly language pgm without macro defn or call.

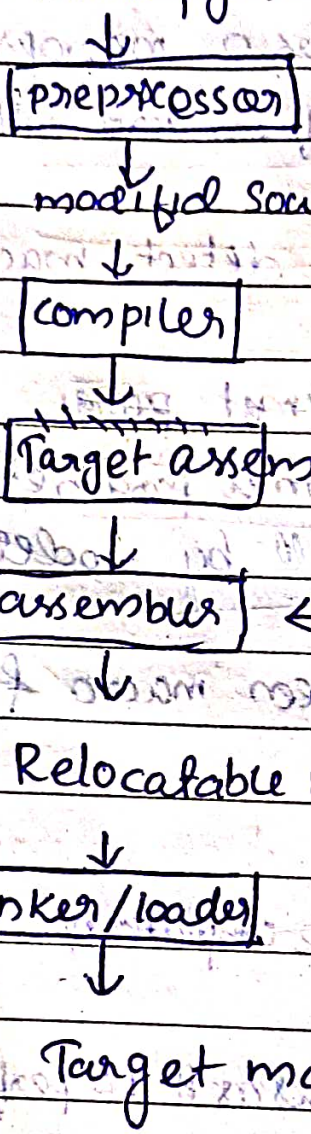


Difference between macro & function with ex

Macros	function
1) preprocessed : Before complation/ assemblying	After assemblying / complation fn is used
2) code length increases as it is performing expansion	→ Same
3) Faster as sequencial execution	→ Slower because of stack
4) Time efficient	→ Not time efficient
5) useful when small code appears in several times	→ use ful when large code appears several time
6) Requires more memory	→ Requires less memory

Page No. :
Date / /

Step for language processing system



Prepocessor - preprocessor is considered as a part of the compiler it is a tool which produces input for compiles - It deals with macroprocessing, augmentation language extensions.

interpreter - An interpreter is like compiler which translates high level language to machine langu

Assembler - It translates assembly lang code into machine code. The output is known as object file. which is a combination of machine instructions.

Linker. The linker helps you link and merge various and object files to create an executable file. All these files might have bee compiled with seperate assembler. the main task of a ~~assembler~~ linker is to search for called modules in a pgm and to find out the memory locations.

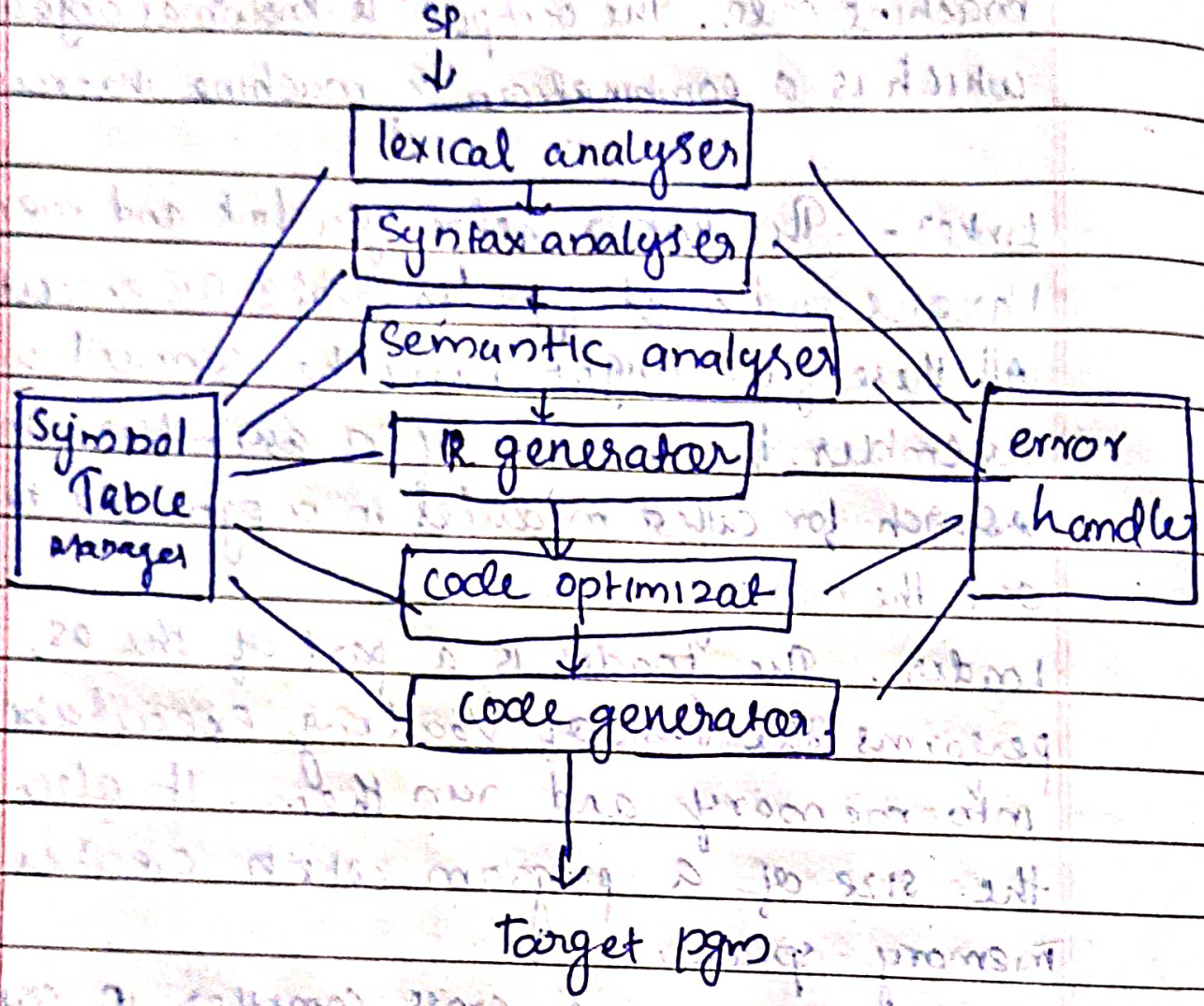
Loader. The loader is a part of the os, which performs the task of loading executable files into memory and run them. It also calculate the size of a program which creates additional memory space.

Cross - Compler - A cross compiler in compiler design is a perform which helps you to generate executable code.

source to source compiler. Source to source compiler is a term used when the source code of one program lang is translated into another lang.

Page No.:
Date / /

compiler the phase of a compiler



Translation of the stmnt position := initial + rate * 60

Position := initial + rate * 60

↓

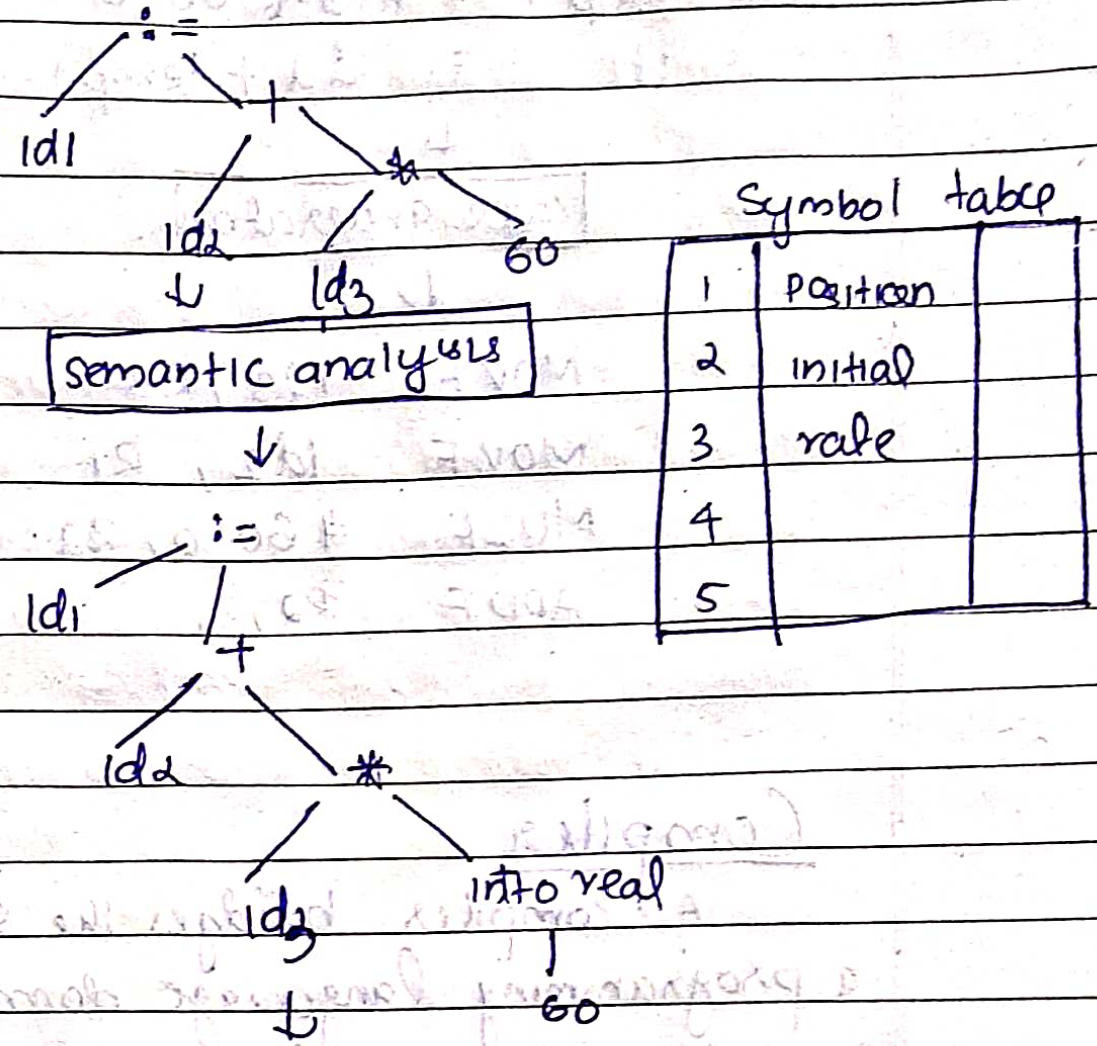
Lexical Analysis

↓

id1 := id2 + id3 * 60

syntax analysis

↓


:=
id1
+
id2
*
id3
60
Symbol table
1 position
2 initial
3 rate
4
5
Semantic analysis
:=
id1
+
id2
*
id3
into real
60
Intermediate representation
temp1 := int to real (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
Code optimization

Page No. :
Date / /

temp 1 := id3 * 60.0
id1 := id2 + temp1

↓

code generation

↓

```
MOVF   id3, R2
MOVF   id2, R1
MULF   #60.0, R2
ADDF   R2, R1
```

Compiler

A compiler bridges the semantic gap b/w a programming language domain and the execution domain by generating a TP, which may be a machine lng prgm or an object module..

Semantic gap is caused by 4 features of the programming language - datatypes, datastructs control structures, scope rules etc.

Compiles pgm performs 2 types of tasks.

- providing diagnostics for violation of PL, Semantic in a sp
- generating code to implement meaning of

a source pgm in the execution domain.

The semantic gap refers to the difference between semantics of 2 domains. A compiler is concerned with the PL domain of its source language and execution domain of its Target machine which is either a host computer or a virtual machine running on some os.

4 features that may bridge semantic gap are

1) datatypes
2) datastructures
3) control structures
4) Scope rules.

Memory allocation:

The life time of a variable is that interval of time during the execution of a pgm over which the variable exists.

Steps involves memory allocation are

1) Determining the amount of memory required for storing a value of data item
2) using an appropriate memory allocation model for implementing the life time of data items and scope rules
3) Developing appropriate memory mapping for accessing

Page No. :
Date / /

Values Stored in a D.S

Static & Dynamic memory allocation

Memory Binding :-

It is an association b/w the memory address attribute of a data item and the address of a memory area.

Memory allocation is the activity of performing memory allocation binding. Memory binding can be static or dynamic

In static memory allocation memory is allocated before execution. It is performed during compilation.

In Dynamic allocation binding are established and destroyed during execution time. hence binding of attributes such as types and dimensions of variables can also be performed during execution

Static allocation - fortran

Dynamic - C, C++, Java.

Dynamic allocation is of 2 types

i) Automatic dynamic allocation

ii) program controlled dynamic allocation

i) Automatic dynamic allocation.

Memory is allocated to the variables in a prgm unit when the prgm unit entered during execution and deallocated when pgm is exited.

ii) program controlled

A prgm can allocate or deallocate memory at arbitary point during its execution. dynamic allocation done by stack or heap.

Compilation of expressions:

The major issues in code generation for expressions are as follows.

- determination of the code in which operators used in an expression should be evaluated.
- choice of instructions for performing an operation
- use of CPU registers and handling of partial result in the generated code.

compilation expression classified as

i) Toy code generation for expression
ii) intermediate code for expression.

Toy code generator

It uses the notion of an operand

descripter to maintain type, length and addressability information for each operand code generator uses a register descripter to maintain info about what operand or particular result would be contained in CPU register during execution of generated code.

The toy code generator for expression is pro passing through following stages

→ Operand descripter
→ register descripter
→ generating information
→ Saving partial results
→ code generator routines
→ expression tree

Operand descripter:

An operand descripter has the following fields

i) Attributes (length and type)
ii) ~~spec~~ Addressibility - specifies where operand is located

Contain 2 subfield

a) Addressability code - M & R
b) Address - address of memory word.

Page No. :
Date / /

M → operand in memory
R ⇒ operand in register.
eg: code generated for expression a+b as follows

```
MOVER AREG, A
MULT  AREG, B
```

These three operand descripter are used during code generation. assuming a, b to be integers occupying 1 memory word, these are

attributes	addressability
(int, 1)	M, addr(a)
(int, 1)	M, addr(b)
(int, 1)	R, addr(AREG

Descriptor for a

Register descriptor:
It has two fields
1) Status: contains code free or occupied to indicate register status.
2) operand descriptor: If Status = occupied this field contains the descriptor for the operand contained in the registers.

Register descriptor are stored in an array indexed Register - ~~class~~ descriptor. one register descriptor exist for each CPU register.

Page No. :
Date / /

Creerating an instruction

when operand op is reduced by the passed, function codegen is called with opi and descriptors of its operands as parameter.

- A single instruction can be generate to evaluate one operand. if descriptor indicate one operand in a register other in memory.

Saving partial results :-

If all registers are occupied when operator opi is to be evaluated, a register r is freed by copying its contents into temporary location. r is now used to evaluate operator. simply an array temp is declared in the TP to hold partical results. A partial result is always stored in ~~memory~~ ~~the~~ in the next free entry of temp. when partial result is moved to temperary location the discriptor of partial result must change. The operand discriptor field ~~is~~ of the regist descriptor is used to achieve this

Page No.:
Date / /

eg : a*b + c*d . after generating code for a*b, operand & register descriptor would as shown. after the partial result a*b is moved to a temporary location temp[i]

attributes	Addressability
(int, 1)	M, addr(a)
(int, 1)	M, addr(b)
(int, 1)	M, addr(temp[i])

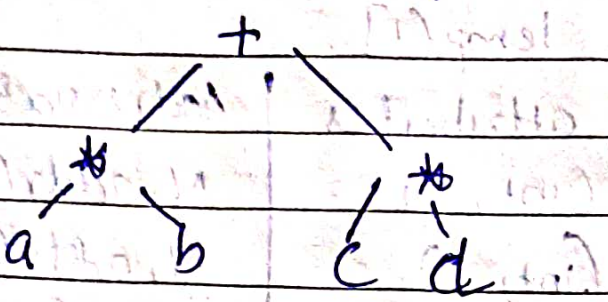
Generation routines

It includes the compute code generation routine. It first checks the addressability of its operands to decide whether one of them exists in a regest. If so, it generates a single instruction to perform the operation. If none of the operands is in a register it need to move one operand into the register before processing. for this purpose it first frees the register if it is occupied and changes the operand descriptor. At the end of the code generation it builds a descriptor for partial result

Page No. :
Date / /

Expression tree

The expression tree for the string (a*b)+(c*d) is as follows



intermediate code for expression
→ postfix string
→ ~~Pre~~ Triples & quadraples
→ Expression trees.

Post fix trees

The postfix notation, each operator appears immediately after its last operand. thus a binary operator op appears after its second operand. Operands can be evaluated in the orders in which they appear in the string
eg: (a*b)+(c*d) the post fix of this string is
step1 ab* +cd*
step2 ab*cd*+

Page No. :
Date / /

Triples & quadruples

A triples is a representation of an elementary operation the form of pseudo machine instruction.

operator	operand 1	operand 2

eg: consider the string a+b*c+d*e↑f

the triple is written as

	operator	operand 1	operand 2
1	*	b	c
2	+	1	a
3	↑	e	f
4	*	2	3
5	+	d	4

A pgm representation called of indirect triples is useful in optimizing the compilers

eg: z := a+b*c+d*e↑f

y := x+b*c

indirect triples shown in below.

Page No. :
Date / /

	operator	operand 1	operand 2
1	*	b	c
2	+	1	a
3	↑	e	f
4	*	d	3
5	+	2	4
6	+	x	

stmt no	discription no
1	1,2,3,4,5
2	1,6

The quadraples represents an elementary evolution in the following format:

operator	operand 1	operand 2	result name

There the result name designated the result of the evaluation :

z := a + b * c + d * e ↑ f

Page No. :
Date 1 . 1

	operator	operand 1	Operand 2	result name
1	*	b	c	t1
2	+	1	a	t2
3	π	e	f	t3
4	*	d	3	t4
5	+	d	4	t5
6				

Expression tree

It is an abstract syntax tree which shows the structure of an expression. It is determined to best evaluation order.

eg :- expression tree for $f+(x+y)*((a+b)/(c-d))$