Chapter 4: Compilers and Interpreters

Mrs. Sunita M Dol (Aher),
Assistant Professor,
Computer Science and Engineering Department,
Walchand Institute of Technology, Solapur, Maharashtra

4. Compilers and Interpreters

- Aspects of compilation
- Static and dynamic memory allocation
- Memory allocation in block structured languages
- Compilation of expressions
- Code Optimization
- Interpreters

4. Compilers and Interpreters

- Aspects of compilation
- Static and dynamic memory allocation
- Memory allocation in block structured languages
- Compilation of expressions
- Code Optimization
- Interpreters

Aspects of Compilation

- A compiler bridges the semantic gap between a PL domain and an execution domain.
- Two aspects of compilations are
  - Generate code to implement meaning of a source program in the execution domain.
  - Provide diagnostics for violations of PL semantics in a source program.

Aspects of Compilation

- PL Features
  - Data Types
  - Data Structures
  - Scope Rules
  - Control Structure

PL Features

- Data Types
  - A data is a specification of
    1. Legal values for variables of the type and
    2. Legal operations on the legal values of the type.
  - The following tasks are involved
    1. Checking legality of an operation for types of operands.
    2. Use type conversion operations.
    3. Use appropriate instruction sequence of the target machine.

PL Features

- Data Types

Example: Consider program segment

```
i: integer;
a, b: real
a := b + I
```

Instruction generated for program segment:

```
CONV_R    AREG, I
ADD_R     AREG, B
MOVEM     AREG, A
```

PL Features

- Data Structure
  - A PL permits the declaration and use of data structure like arrays, stacks, records, lists etc.
  - To compile a reference to an element of a data structure, compiler must develop a memory mappings to access memory word(s) allocated to the element.
  - A user defined type requires mapping of a different kind.

PL Features

- Data Structure Example:

```
program example (input, output);
   type
            employee = record
                        name : array [1..10] of character;
                        sex : character;
                        id : integer;
            end
            weekday = (mon, tue, wed, thu, fri, sat);
   var
            info : array [1..500] of employee;
            today : weekday;
            i, j : integer;
   begin
            today := mon;
            info[i].id := j;
            if today = tue then…
end
```

PL Features

- Scope Rule
  - It determines the accessibility of variables declared in different block of a program.
  - Example:

```
        x, y : real;
             y, z: integer;
A     B             x := y
```

PL Features

- Control structure
  - The control structure of a language is the collection of language features for altering the flow of control during the execution of a program.
  - Example:

```
for i := 1 to 100 do
begin
        lab1: if i = 10 then..
end;
```

4. Compilers and Interpreters

- Aspects of compilation
- Static and dynamic memory allocation
- Memory allocation in block structured languages
- Compilation of expressions
- Code Optimization
- Interpreters

Memory Allocation

- Memory allocation involves three important task.
  - Determine the amount of memory required to represent the values of a data item.
  - Use an appropriate memory allocation model to implement the lifetimes and scope of data items.
  - Determine appropriate memory mappings to access the values in a non-scalar data item.

Static and Dynamic Memory Allocation

- Memory Binding is an association between the 'memory address' attribute of a data item and the address of a memory area.
  - In static memory allocation, memory is allocated to a variable before the execution of program begins.
  - In dynamic memory allocation, memory bindings are established and destroyed during the execution of a program.

Static and Dynamic Memory Allocation

Example:

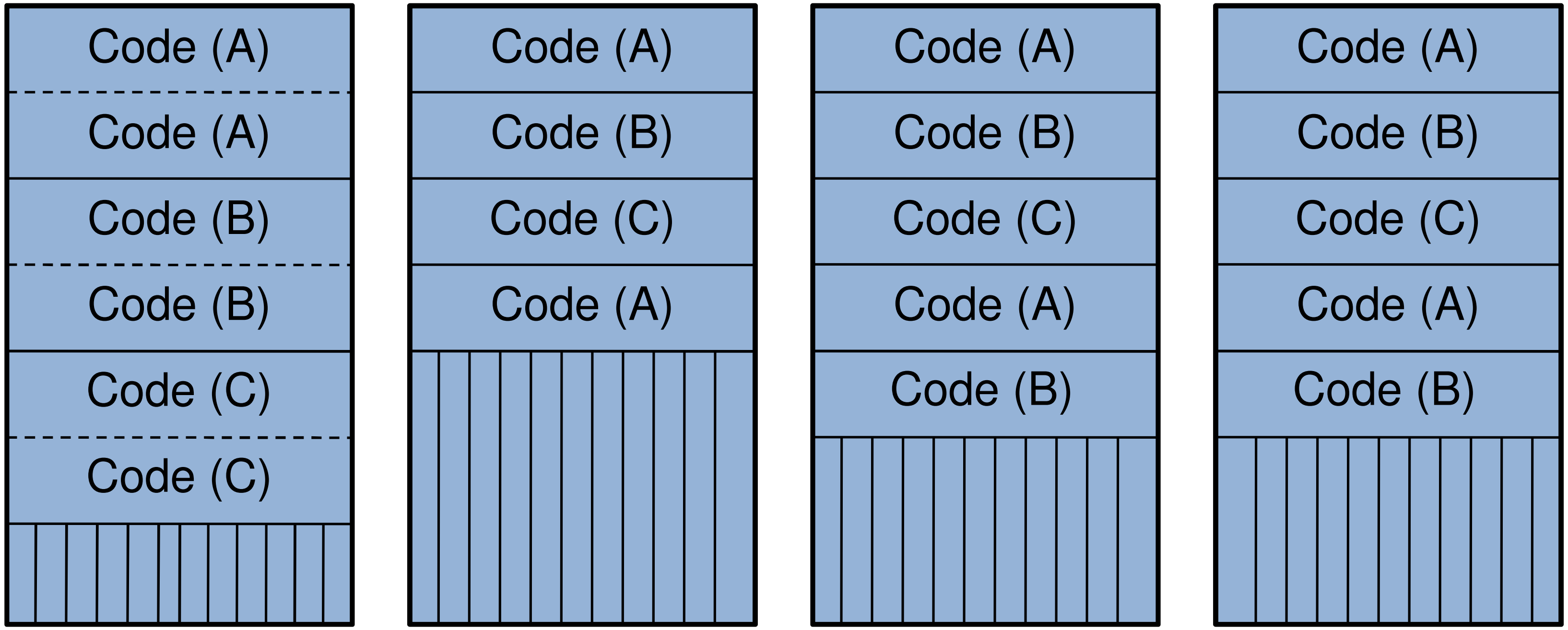


Fig a: Static memory allocation

Fig b: Dynamic memory allocation when only program unit A is active

Fig c: the situation after A calls B

Fig d: the situation after B returns to A and A calls C

Static and Dynamic Memory Allocation

- Dynamic memory allocation has two flavours:
  - Automatic allocation: memory is allocated to the variables in a program unit when the program unit is entered during execution and is deallocated when the program is exited.
  - Program controlled allocation: a program can allocate or deallocate memory at arbitrary points during its execution.

Static and Dynamic Memory Allocation

- Dynamic memory allocation can be implemented using stack and heaps.
  - Automatic dynamic allocation is implemented using stack.
  - Program controlled allocation is implemented using heap.

Static and Dynamic Memory Allocation

- Dynamic memory allocation advantages
  - Recursion can be implemented easily.
  - Dynamic allocation can support data structure whose sizes are determined dynamically e.g. Array.

4. Compilers and Interpreters

- Aspects of compilation
- Static and dynamic memory allocation
- Memory allocation in block structured languages
- Compilation of expressions
- Code Optimization
- Interpreters

Memory Allocation in Block Structured Language

- Scope Rules
  - A data declaration using a name *$name_i$* creates a variable *var_i*. *($name_i$, var_i)*
  - Variable *var_i* is visible at a place in the program if some binding *($name_l$, var_i)* is effective at that place.
  - It is possible for data declaration in many blocks of a program to use a same name say *$name_i$*.
  - Scope rules determine which of these bindings is effective at a specific place in the program.

Memory Allocation in Block Structured Language

- Scope of a variable
  - If a variable var_i is created with the name $name_i$ in a block b,
    - var_i can be accessed in any statement situated in block b
    - var_i can be accessed in any statement situated in a block b’ which is enclosed in b unless b’ contains a declaration using the same name.
  - A variable declared in block b is called a local variable of block b.
  - A variable of enclosing block that is accessible within block b is called a nonlocal variable.

Memory Allocation in Block Structured Language

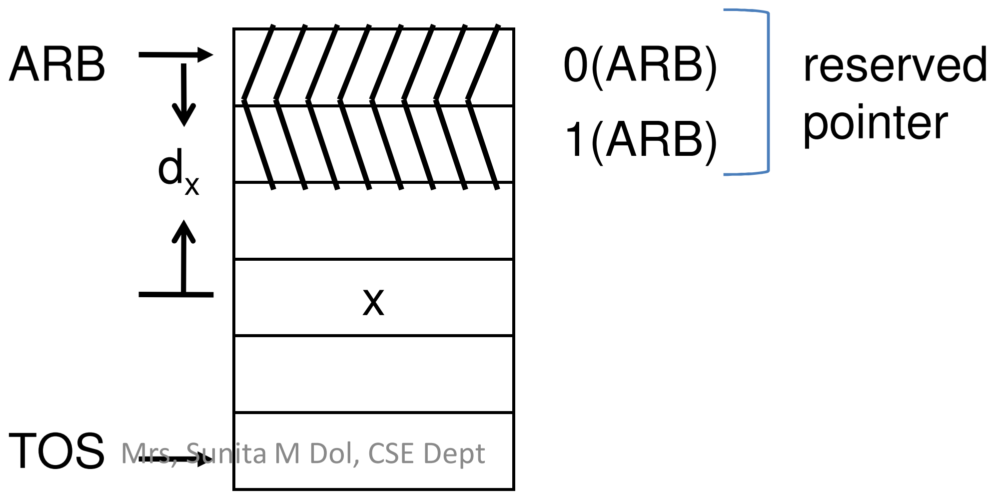
- Scope of a variable
  - Example

```
A [ x, y, z : integer;
  B [ g : real;
    C [ h, z : real;
  D [ i, j : integer;
```

Block	Accessible variables	
	local	nonlocal
A	x_A, y_A, z_A	---
B	g_B	x_A, y_A, z_A
C	h_C, z_C	x_A, y_A, g_B
D	i_D, j_D	x_A, y_A, z_A

Memory Allocation in Block Structured Language

- Memory allocation and access
  - Automatic memory allocation can be implemented using the extended stack model.
  - Each record in the stack has two reserved pointers.
  - Each stack record accommodates the variables for one activation of a block, hence called an activation record.



Memory Allocation in Block Structured Language

- Memory allocation and access
  - Example

```
A  B  C
x, : real;
   y : char;
      z, w : integer;
      z:=10;
      x:=z;
```

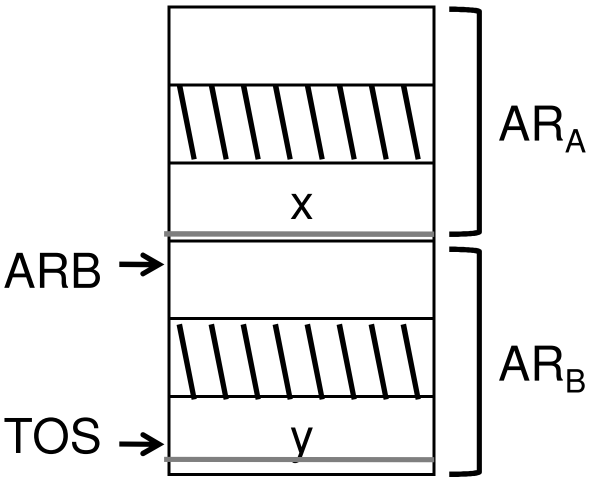


Fig 1: situation when block A and B are active

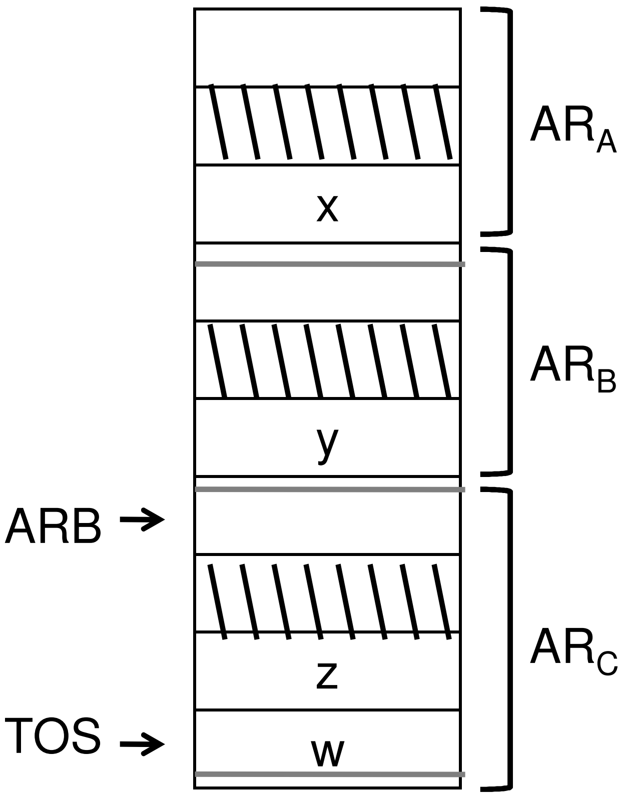


Fig 2: situation after entry to c

Memory Allocation in Block Structured Language

- Memory allocation and access
  - Action at entry of block C

Sr. No.	Statement
1	TOS := TOS + 1;
2	TOS* := ARB; {set the dynamic pointer}
3	ARB := TOS;
4	TOS := TOS + 1;
5	TOS* := …; {set the dynamic pointer 2}
6	TOS := TOS + n;

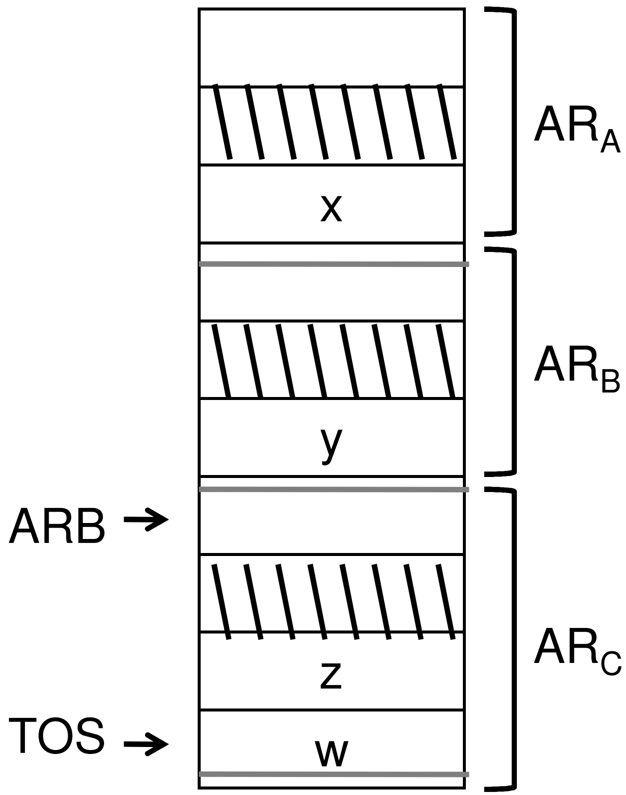


Fig 2: situation after entry to c

Memory Allocation in Block Structured Language

- Memory allocation and access
  - Action at exit of block C

Sr. No.	Statement
1	TOS := ARB - 1;
2	ARB := ARB*;

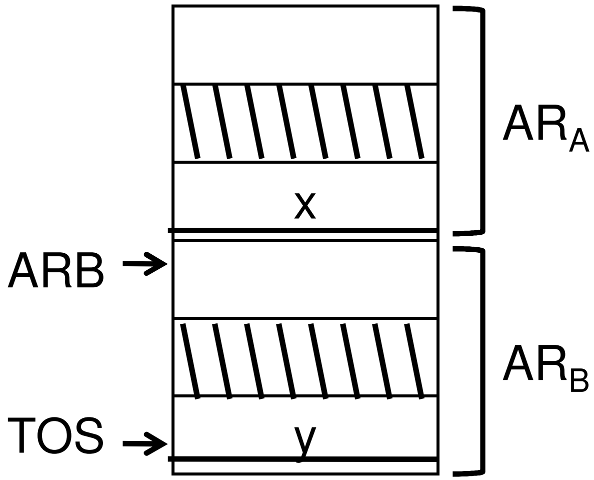


Fig 1: situation when block A and B are active

4. Compilers and Interpreters

- Aspects of compilation
- Static and dynamic memory allocation
- Memory allocation in block structured languages
- Compilation of expressions
- Code Optimization
- Interpreters

Compilation of expressions

- A Toy Code Generator for Expression
- Intermediate Codes for Expression

A Toy Code Generator

- A Toy Code Generator for Expression

The major issues in code generation for expression are:

  - Determination of an evaluation order for the operators in an expression
    - Top down and bottom up parse
  - Selection of instruction to be used in the target code.
    - Operand Descriptor
  - Use of registers and handling of partial result.
    - Register Descriptor

A Toy Code Generator

- Operand Descriptor

  An operand descriptor has the following fields:

  – Attributes : contains subfields type, length and miscellaneous information

  – Addressability

    - Addressability Code- M, R, AM, AR
    - Address

  Example:

MOVER AREG, A
MULT AREG, B

(int, 1)	M, addr(a)	Descriptor for a
(int, 1)	M, addr(b)	Descriptor for b
(int, 1)	R, addr(AREG)	Descriptor for a*b

A Toy Code Generator

- Register Descriptor

An register descriptor has the following fields:

– Status: *occupied* or *free*

– Operand Descriptor#

Example:

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

Operand Descriptor

(int, 1)	M, addr(a)	Descriptor for a
(int, 1)	M, addr(b)	Descriptor for b
(int, 1)	R, addr(AREG)	Descriptor for a*b

Register Descriptor

Occupied	#3

A Toy Code Generator

- Generating Instruction
  - Single instruction can be generated to evaluate op_i if the descriptor indicate that one operand is in a register and the other is in memory.
  - If both operands are in memory, an instruction is generated to move one of them into a register. This is followed by an instruction to evaluate op_i.

A Toy Code Generator

- Saving Partial Result
  - If all registers are occupied when operator op_i is to be evaluated, a register r is freed by copying its contents into a temporary location in the memory.

Example: Consider the expression a*b + c*d. After generating code for a*b, the operand and register descriptor would be as shown below.

Operand Descriptor

(int, 1)	M, addr(a)	Descriptor for a
(int, 1)	M, addr(b)	Descriptor for b
(int, 1)	R, addr(AREG)	Descriptor for a*b

Register Descriptor

Occupied	#3

After the partial result a*b moved to a temporary location, the operand descriptor must become:

(int, 1)	M, addr(a)	Descriptor for a
(int, 1)	M, addr(b)	Descriptor for b
(int, 1)	M, addr(temp[1])	Descriptor for a*b

A Toy Code Generator

- Code Generation Routine

```
codegen(operator, opd1, opd2)
{
        if opd1.addressability_code = ‘R’
                if operator = ‘+’ generate ‘ADD AREG, opd2’;
        else if opd2.addressability_code = ‘R’
                if operator = ‘+’ generate ‘ADD AREG, opd1’;
        else
                if Register_descr.status = ‘occupied’
                        generate (‘MOVEM AREG, Temp[j]’);
                        j = j + 1;
                        Operand_descr [Register_descr.Operand_descriptor#] =
                           (<key>, (M, Addr(Temp[j])));
                        generate ‘MOVEM AREG, opd1’
                if operator = ‘+’ generate ‘ADD AREG, opd2’;
        i = i + 1;
        Operand_descr[i] =  (<key>, (R, Addr(AREG)));
        Register_descr[i] =  (‘Occupied’, i);
        return I;
}
```

A Toy Code Generator

- Code Generation actions for a*b + c*d

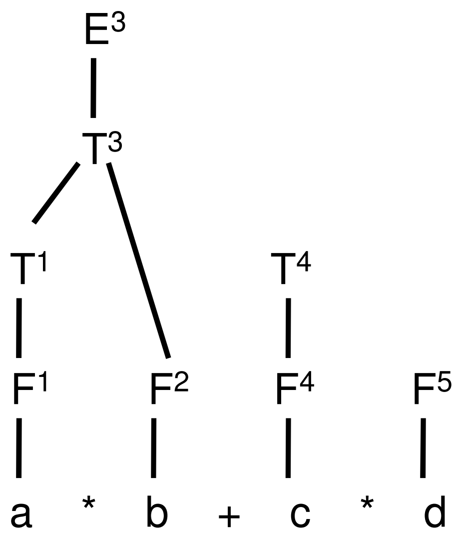
Step No.	Parsing action	Code generation action
1	$\langle id \rangle_a \rightarrow F^1$	Build descriptor # 1
2	$F^1 \rightarrow T^1$	-
3	$\langle id \rangle_b \rightarrow F^2$	Build descriptor # 2
4	$T^1 * F^2 \rightarrow T^3$	Generate MOVER AREG, A MULT AREG, B Build descriptor # 3
5	$T^3 \rightarrow E^3$	-
6	$\langle id \rangle_c \rightarrow F^4$	Build descriptor # 4
7	$F^4 \rightarrow T^4$	-
8	$\langle id \rangle_d \rightarrow F^5$	Build descriptor # 5

9	$T^4 * F^5 \rightarrow T^6$	Generate MOVEM AREG, TEMP_I
		MOVER AREG, C
		MULT AREG, D
		Build descriptor # 6

10	$E^3 + T^6 \rightarrow E^7$	Generate ADD AREG, TEMP_I

A Toy Code Generator

- Code Generation actions for a*b + c*d



Operand Descriptor

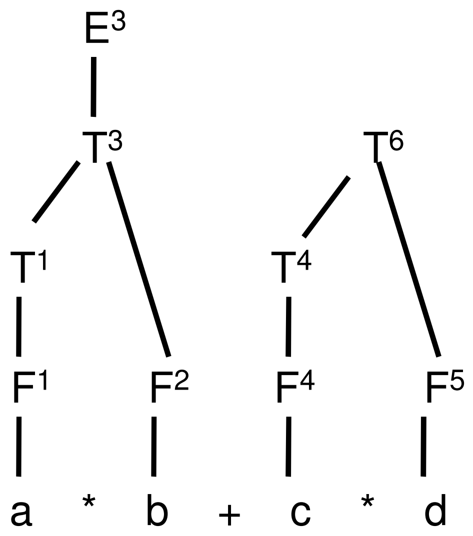
1	(int, 1)	M, addr(a)	Descriptor for a
2	(int, 1)	M, addr(b)	Descriptor for b
3	(int, 1)	R, addr(AREG)	Descriptor for a*b
4	(int, 1)	M, addr(c)	Descriptor for c
5	(int, 1)	M, addr(d)	Descriptor for d

Register Descriptor

Occupied	#3

A Toy Code Generator

- Code Generation actions for a*b + c*d



Operand Descriptor

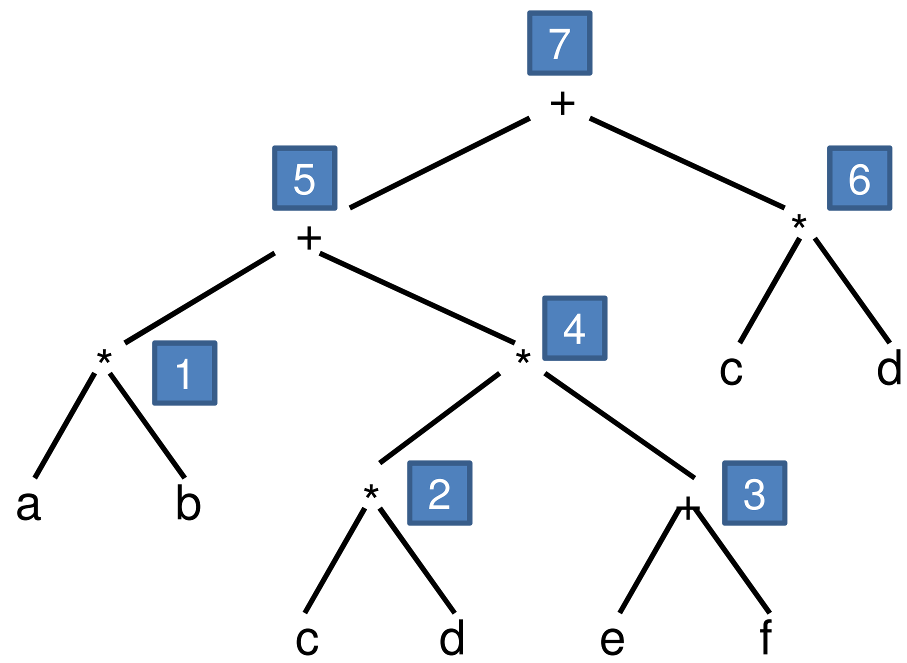
1	(int, 1)	M, addr(a)	Descriptor for a
2	(int, 1)	M, addr(b)	Descriptor for b
3	(int, 1)	M, addr(temp[1])	Descriptor for a*b
4	(int, 1)	M, addr(c)	Descriptor for c
5	(int, 1)	M, addr(d)	Descriptor for d
6	(int, 1)	R, addr(AREG)	

Register Descriptor

Occupied	#6

A Toy Code Generator

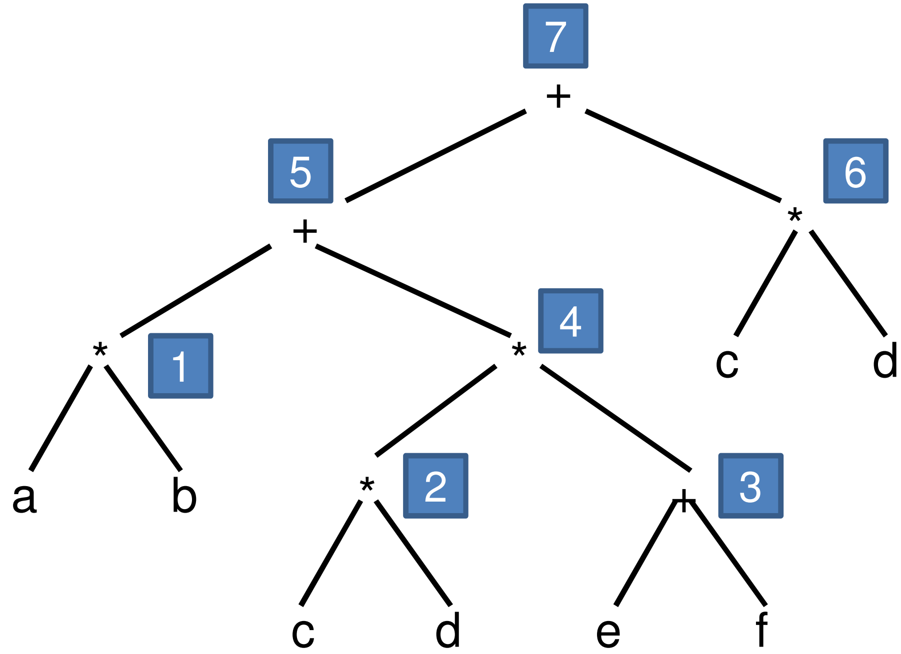
- Expression tree for a*b+c*d*(e+f)+c*d



A Toy Code Generator

- Expression tree for a*b+c*d*(e+f)+c*d

```
MOVER   AREG,   A
MULT    AREG,   B
MOVEM   AREG,   TEMP_1
MOVER   AREG,   C
MULT    AREG,   D
MOVEM   AREG,   TEMP_2
MOVER   AREG,   E
ADD     AREG,   F
MULT    AREG,   TEMP_2
ADD     AREG,   TEMP_1
MOVEM   AREG,   TEMP_3
MOVER   AREG,   C
MULT    AREG,   D
ADD     AREG,   TEMP_3
```

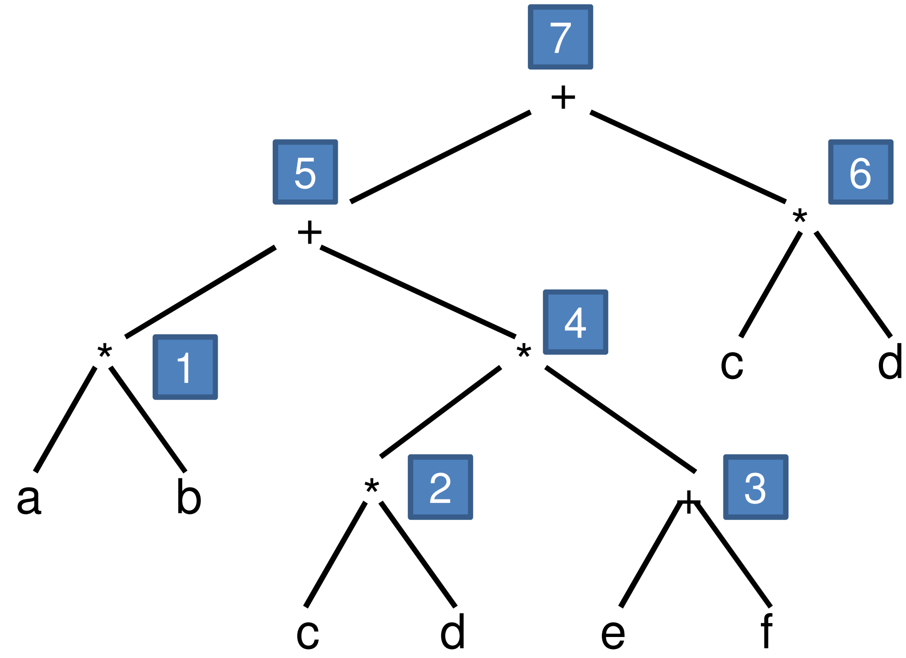


Temporary	Contents	Used in
temp[1]	Value of node 1	Evaluating node 5
temp[2]	Value of node 2	Evaluating node 4
temp[3]	Value of node 5	Evaluating node 7

A Toy Code Generator

- Expression tree for a*b+c*d*(e+f)+c*d

```
MOVER   AREG,   A
MULT    AREG,   B
MOVEM   AREG,   TEMP_1
MOVER   AREG,   C
MULT    AREG,   D
MOVEM   AREG,   TEMP_2
MOVER   AREG,   E
ADD     AREG,   F
MULT    AREG,   TEMP_2
ADD     AREG,   TEMP_1
MOVEM   AREG,   TEMP_1
MOVER   AREG,   C
MULT    AREG,   D
ADD     AREG,   TEMP_1
```



Temporary	Contents	Used in
temp[1]	Value of node 1	Evaluating node 5
temp[2]	Value of node 2	Evaluating node 4
Temp[1]	Value of node 5	Evaluating node 7

Compilation of expressions

- A Toy Code Generator for Expression
- Intermediate Codes for Expression

Compilation of expressions

- Intermediate Codes for Expression
  a. Postfix strings
  b. Triples and quadruples
  c. Expression trees

Intermediate Code for Expressions

- Postfix Strings
  - Each operator appears immediately after its last operand.
  - Example

$$\vdash\ a \overset{\boxed{2}}{+} b \overset{\boxed{1}}{*} c \overset{\boxed{5}}{+} d \overset{\boxed{4}}{*} e \overset{\boxed{3}}{\uparrow} f\ \dashv$$

$$\text{Postfix } \vdash\ a\ b\ c\ \overset{\boxed{1}}{*}\ \overset{\boxed{2}}{+}\ d\ e\ f\ \overset{\boxed{3}}{\uparrow}\ \overset{\boxed{4}}{*}\ \overset{\boxed{5}}{+}\ \dashv$$

Intermediate Code for Expressions

- Postfix Strings
  - Stack of operand descriptor can be used to perform code generation.
    - Operand descriptors are pushed on the stack as operands appear in the string.
    - When a operand with arity k appears in the string, k descriptors are popped off the stack.
    - A descriptor for the partial result generated by the operator is now pushed on the stack.

Intermediate Code for Expressions

- Triples and quadruples
  - Triple is a representation of an elementary operation in the form of a pseudo-machine instruction.

Operator	Operand1	Operand2

  - Example: Triples for a + b * c + d * e ↑ f

	Operator	Operand1	Operand2
1	*	b	c
2	+	1	a
3	↑	e	f
4	*	d	3
5	+	2	4

Intermediate Code for Expressions

- Triples and quadruples
  - In indirect triple, a table is built to contain all distinct triples in the program. A program statement is represented as a list of triple numbers.
  - Example: Triples for $x = a + b * c + d * e \uparrow f$

    & $y = x + b * c$

	Operator	Operand1	Operand2
1	*	b	c
2	+	1	a
3	↑	e	f
4	*	d	3
5	+	2	4
6	+	x	1

Statement no.	Triple no.
1	1, 2, 3, 4, 5
2	1, 6

Intermediate Code for Expressions

- Triples and quadruples
  - Quadruple represents an elementary evaluation in the following format

Operator	Operand1	Operand2	Result name

  - Example: Triples for a + b * c + d * e ↑ f

	Operator	Operand1	Operand2	Result name
1	*	b	c	t1
2	+	1	a	t2
3	↑	e	f	t3
4	*	d	3	t4
5	+	2	4	t5

Intermediate Code for Expressions

- Expression Trees
  - It is an abstract syntax tree which depicts the structure of an expression.
  - It is used to determine the best evaluation order.
  - Example (a + b) / (c + d)

MOVER	AREG,	A	MOVER	AREG,	C
ADD	AREG,	B	ADD	AREG,	D
MOVEM	AREG,	TEMP_1	MOVEM	AREG,	TEMP_1
MOVER	AREG,	C	MOVER	AREG,	A
ADD	AREG,	D	ADD	AREG,	B
MOVEM	AREG,	TEMP_2	DIV	AREG,	TEMP_1
MOVER	AREG,	TEMP_1			
DIV	AREG,	TEMP_2			

Figure: Alternative Code for (a + b) / (c + d)

Intermediate Code for Expressions

- Expression Trees
  - Two step procedure is used to determine the best evaluation order for the operation in an expression.
  - The first step associates a register requirement label with each node in the expression.
  - The second step analyses the RR labels of the child nodes of s node to determine the order in which they should be evaluated.

Intermediate Code for Expressions

- Expression Trees

Algorithm – Evaluation order for operators

1. Visit all nodes in an expression tree in post order.

   For each node n_i

   (a) if n_i is a leaf node then

   if n_i is the left operand of its parent then

   $RR(n_i) := 1;$

   else $RR(n_i) := 0;$

   (b) if ni is not a leaf node then

   if $RR(l\_child_{ni}\ )$ NE $RR(r\_child_{ni})$ then

   $RR(n_i) := max(RR(r\_child_{ni}), RR(l\_child_{ni});$

   else $RR(n_i) := RR(l\_child_{ni}) + 1;$

Intermediate Code for Expressions

- Expression Trees

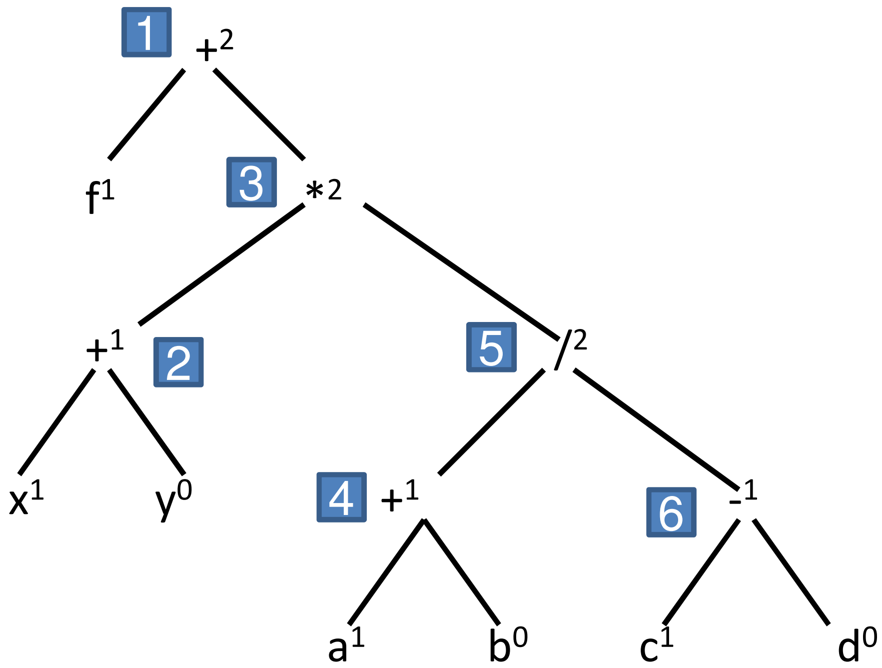
Algorithm – Evaluation order for operators

2. Perform the procedure call evaluation_order (root) which prints a postfix form of the source string in which operators appear in the desired evaluation order.

```
Procedure evaluation_order (node);
        if node is not a leaf node then
                if RR(l_child_node) ≤ RR(r_child_node )
                        evaluation_order(r_child_node);
                        evaluation_order(l_child_node);
                else
                        evaluation_order(l_child_node);
                        evaluation_order(r_child_node);
        print node;
end evaluation_order;
```

Intermediate Code for Expressions

- Expression Trees Example:
  - Expression tree for f + (x + y) * ((a + b) / (c – d))



4. Compilers and Interpreters

- Aspects of compilation
- Static and dynamic memory allocation
- Memory allocation in block structured languages
- Compilation of expressions
- Code Optimization
- Interpreters

4. Compilers and Interpreters

- Aspects of compilation
- Static and dynamic memory allocation
- Memory allocation in block structured languages
- Compilation of expressions
- Code Optimization
- Interpreters

Code Optimization

- Code optimization aims at improving the execution efficiency of a program.
- This is achieved in two ways:
  – Redundancies in a program are eliminated.
  – Computations in a program are rearranged to make it executes efficiently.

Code Optimization

- Code optimization

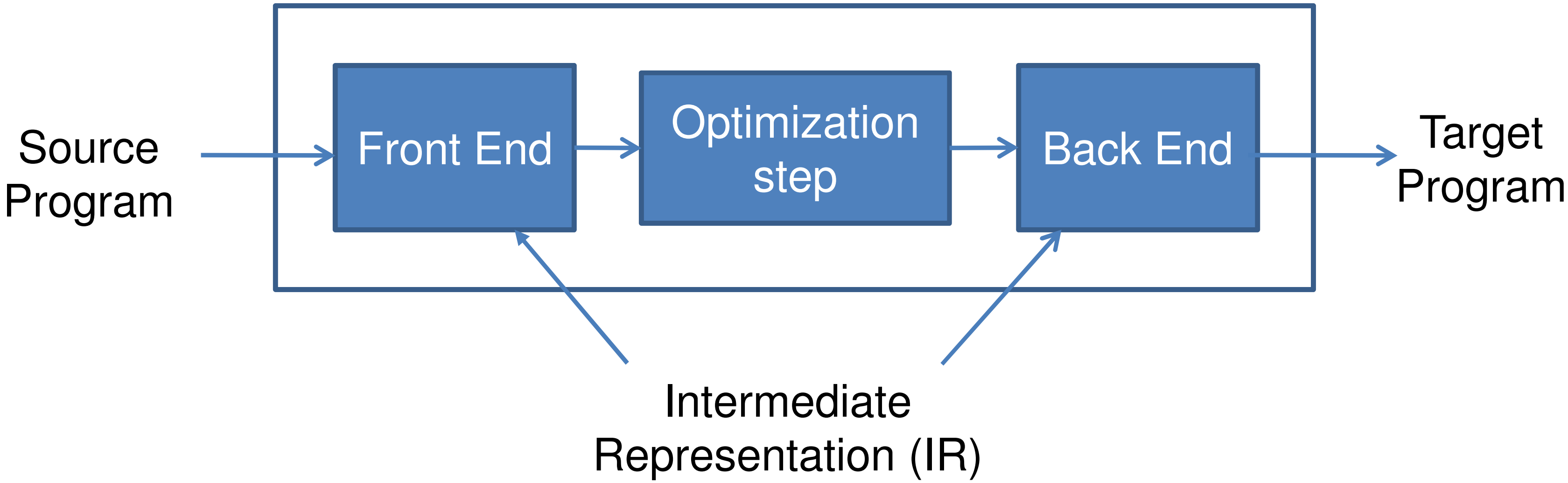


Figure: Schematic of an optimizing compiler

Code Optimization

- Optimizing Transformation
  - An optimizing transformation is a rule for rewriting a segment of a program to improve its execution efficiency without affecting its meaning.
  - Optimizing transformations used in compilers
    - Compile time evaluation
    - Elimination of common subexpression
    - Dead code elimination
    - Frequency reduction
    - Strength reduction

Code Optimization

- Optimizing Transformation
  - Compile time evaluation:
    - Execution efficiency can be improved by performing certain actions specified in a program during compilation itself.
    - Constant folding is the main optimization of this kind.

      e.g. an assignment a:= 3.14157/2 can be replaced by a:= 1.570785

Code Optimization

- Optimizing Transformation
  - Elimination of common subexpression
    - Common subexpressions are occurrences of expression yielding the same value.

      e.g.

```
a := b * c;
-----
x := b * c + 5.2;
```

After eliminating the common subexpression

```
t := b * c;
a := t;
-----
x := t + 5.2;
```

Code Optimization

- Optimizing Transformation
  - Dead code elimination
    - Code which can be omitted from a program without affecting its result is called dead code.

      e.g. an assignment x := <exp> constitute dead code if the value assigned to it is not used in the program, no matter how control flows after executing this assignment.

Code Optimization

- Optimizing Transformation
  - Frequency reduction
    - Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times.

```
for I := 1 to 100 do
begin
        z := i;
        x := 25 * a;
        y := x + z;
end;
```

```
x := 25 * a;
for I := 1 to 100 do
begin
        z := i;
        x := 25 * a;
        y := x + z;
end;
```

Code Optimization

- Optimizing Transformation
  - Strength reduction
    - Strength reduction optimization replaces the occurrences of a time consuming operation by an occurrence of a faster operation.
    - e.g. replacement of a multiplication by an addition.

```
for I := 1 to 10 do
begin
        --------
        k := i * 5;
        ---------
end;
```

```
itemp := 5;
for I := 1 to 10 do
begin
        --------
        k := itemp
        ---------
        itemp := itemp + 5;
end;
```

Code Optimization

- Optimizing Transformation
  - Strength reduction
    - Strength reduction is very important for array access within program loops.
    - Strength reduction optimization is not performed on operations involving floating point operands.

Code Optimization

- Local and global optimization
  - Optimization of a program is structured into the following two phases
    - **Local optimization** : this optimization are applied over small segments of a program consisting of a few statements.
    - **Global optimization:** this optimization are applied over a program unit i.e. over a function or procedure.

Code Optimization

- Local optimization
  - Scope of local optimization is a basic block.
  - A basic block is a sequence of program statements (S_1, S_2, …. , S_n) such that only S_n can be a transfer of control statement and only S_1 can be the destination of a transfer of control statement.

Code Optimization

- Local optimization
  – e.g.

```
          a := x * y;
          --------
          b := x * y;
lab_i:    c := x * y;
```

⇒

```
          t := x * y;
          a := t;
          --------
          b := t;
lab_i:    c := x * y;
```

  – Local optimization identifies two basic blocks in the program
    - The first block extends up to the statement b := x * y.
    - The second basic contains the statement

      lab_i : c := x * y;

Code Optimization

- Local optimization
  - Value number
    - Value numbers are used to determine if two occurrences of an expression in a basic block are equivalent.
    - The value number vn_{alpha} is associated with variable *alpha*
    - If statement n, the current statement being processed, is an assignment to *alpha*.

Code Optimization

- Local optimization
  - Value number
    - A new field is added to each symbol table entry to hold the value number of a variable.
    - The IC for a basic block is a list of quadruples stored in a tabular form
    - A boolean flag *save* is associated with each quadruple to indicate whether its value should be saved for use elsewhere in the program.

Code Optimization

- Local optimization
  - Value number example

Stmt no.	Statement
14	g := 25.2;
15	x := z + 2;
16	h := x * y + d;
..	....
34	w := x * y

Symbol table

Symbol	….	Value number
y		0
x		15
g		14
z		0
d		5
w		0

Code Optimization

- Local optimization
  - Value number example

Quadruple table

	operator	Operand 1		Operand2		Result name	Use flag
		operand	Value no.	Operand	Value no.		
20	:=	g	--	25.2	--	t20	*f*
21	+	z	0	2	--	t21	*f*
22	:=	x	0	t21	--	t22	*f*
23	*	x	15	y	0	t23	~~*f*~~ *t*
24	+	t23	--	d	5	t24	*f*
	..						
57	:=	w	w	t23	--	t57	*f*

Code Optimization

- Global optimization
  - Global optimization requires more analysis effort .
  - Global common subexpression elimination: if some expression x * y occurs in a set of basic block SB of program P, its occurrence in a block b_j can eliminated if following two conditions are satisfied for ever execution of P:
    - x * y is evaluated before execution reaches block b_j and
    - The evaluated value is equivalent to value of x * y in block b_j

Code Optimization

- Global optimization
  - Program representation: a program flow graph for a program P is a directed graph $G_P = (N, E, n_0)$

    where N : set of basic block in P

    E : set of directed edges (b_i, b_j) indicating the possibility of control flow from the last statement b_i to the first statement of b_j

    n_0 : start node of P

Code Optimization

- Global optimization
  - Control flow analysis
    - control flow analysis analyses a program to collect information concerning its structure e.g. presence and nesting of loops in the program.

Code Optimization

- Global optimization
  - Control flow analysis
    - The control flow concepts of interest are:
      - Predecessor and successor
      - Paths
      - Ancestors and predecessor
      - Dominators and post-dominators

Code Optimization

- Global optimization
  - Data flow analysis

Data flow concept	Optimization in which used
Available expression	Common subexpression elimination
Live variable	Dead code elimination
Reaching definition	Constants and variable propagation

Code Optimization

- Global optimization
  - Data flow analysis:
    - This technique analyses the use of data in a program to collect information for the purpose of optimization.
    - This information is computed at entry and exit of each basic block.

Code Optimization

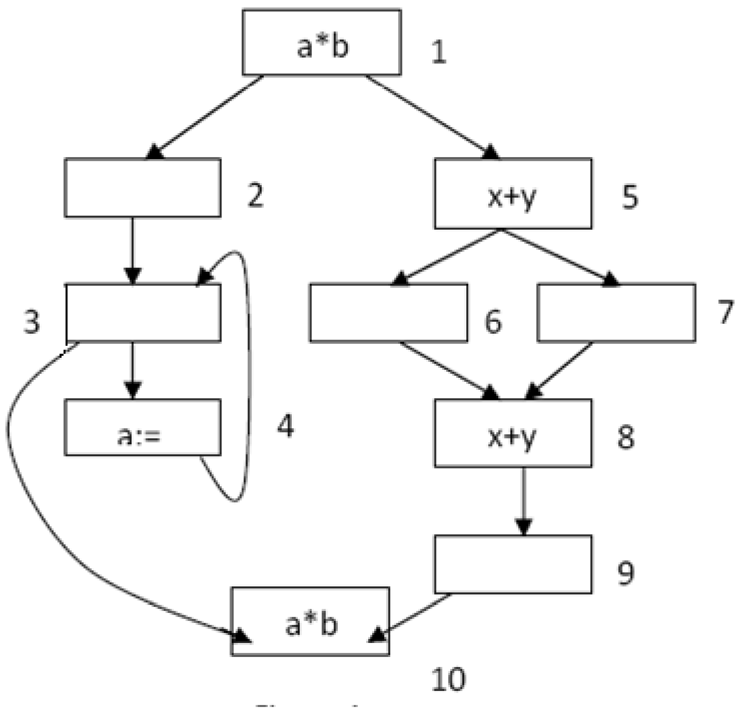
- Global optimization
  - Data flow analysis:
    - Available information: the availability of information at entry or exit of basic block b_i is computed using following rules
      1. Expression e is available at the exit of b_i if
         i. b_i contains an evaluation of e which is not followed by assignment to any operands of e or
         ii. The value of e is available at the entry to b_i and b_i does not contain assignment to any operands of e.
      2. Expression e is available at entry to b_i if it is available at the exit of each predecessor of b_i in program flow graph

Code Optimization

- Global optimization
  - Data flow analysis:
    - Available information
      - The boolean variables avail_in and avail_out are used to represent the availability of expression e at entry and exit of basic block b_i respectively.

Code Optimization

- Global optimization
  - Data flow analysis:
    - Available information



a*b
avail_in = true for blocks 2, 5, 6, 7, 8, 9
avail_out = true for blocks 1, 2, 5, 6, 7, 8, 9, 10

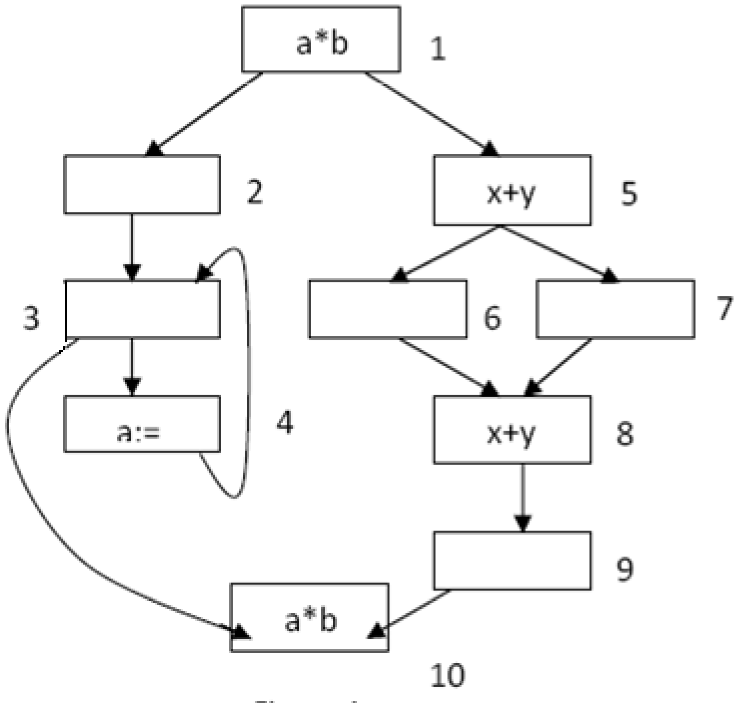
x+y
avail_in = true for blocks 6, 7, 8, 9
avail_out = true for blocks 5, 6, 7, 8, 9

Code Optimization

- Global optimization
  - Data flow analysis:
    - Live variable: the liveness property of a variable can be determined as follows:
      1. Variable v is available at the entry of b_i if
         i. b_i contains a use of e which is not preceded by assignment to v or
         ii. v is live at the exit of b_i and b_i does not contain assignment to v.
      2. v is is available at exit to b_i if it is live at the entry of some successor of b_i in program flow graph

Code Optimization

- Global optimization
  - Data flow analysis:
    - Live variable



Variable b is live at the entry of all blocks,
a is live at the entry of all blocks excepting block 4
and variables x, y are live at the entry of blocks
1, 5, 6, 7 and 8.

Interpreters

- Interpretation avoids the overheads of compilation.
- Interpretation is expensive in terms of CPU time.
- For comparison of compilers and interpreters, consider the following notations:
  - tc : average compilation time per statement
  - te : average execution time per statement
  - ti : average interpretation time per statement

Interpreters

- Both compiler and interpreter analyze a source statement to determine its meaning.
- During compilation, analysis of a statement is followed by code generation and during interpretation, it is followed by actions which implements its meaning.
- tc = ti*te
- Let's assume that tc = 20 te.
- Consider a program P.
- Let $size_P$: number of statement in P

  $stmts\_executed_P$: number of statement executed in some execution of P

Interpreters

- Example

Let $size_P$ = 200. For a specific set of data, let program P execute as follows:

  - 20 statements are executed for initialization purpose.
  - This is followed by 10 iterations of a loop containing statement followed by
  - the execution of 20 statements for printing result.

✓ stmts_executedP = 20 + 10 * 8 + 20
= 120

✓ Total execution time using the compilation model = 200.tc + 120.te
≅ 206.tc

✓ Total execution time using interpretation model = 120.ti

Interpreters

- Use of interpreters
  - Use of interpreters are motivated by two reasons:
    - Efficiency in certain environments
    - simplicity
  - It is simpler to develop an interpreter than to develop a compiler.
  - Interpreter is used in situations where programs or commands are not executed repeatedly.

Interpreters

- Overview of Interpretation
  - Interpretation schematic implements the meaning of a statement without generating code for it.
  - The interpreter consist of three main components:
    - **Symbol table**: holds information concerning entities in the source program.
    - **Data Store**: contains values of the data items declared in the program being interpreted. It consist of a set of component $\{comp_i\}$
    - **Data Manipulation routines**: A set of data manipulation routines contains a routine for every legal data manipulation action in the source language

Interpreters

- Overview of Interpretation
  - Example: the meaning of the statement a:= b + c where a, b, c are of the same type can be implemented by executing the calls

    add(b, c, result)

    assign(a, result)

Interpreters

- Overview of Interpretation
  - Advantages
    - The meaning of the source statement is implemented through execution of the interpreter routine rather than through code generation.
    - Avoiding generation of machine language instruction helps to make the interpreter portable.

Interpreters

- Toy Interpreter
  - Design and operation of an interpreter for Basic written in Pascal.
  - The data store of interpreter consist of two large array
    - rvar to store real
    - ivar to store integer
  - Last few locations in the rvar and ivar arrays are used as stacks for expression evaluation with the help of the pointers r_tos and i_tos respectively.

Interpreters

- Toy Interpreter
  - Design and operation of an interpreter for Basic written in Pascal.
  - The data store of interpreter consist of two large array
    - rvar to store real
    - ivar to store integer
  - Last few locations in the rvar and ivar arrays are used as stacks for expression evaluation with the help of the pointers r_tos and i_tos respectively.

Interpreters

- Basic Interpreter written in Pascal

```
program interpreter (source, output)
    type
        symentry = record
                symbol : array [1..10] of character;
                type : character;
                address : integer;
        end
    var
        symtab : array [1..100] of symentry;
        rvar : array [1..100] of real;
        ivar : array [1..100] of integer;
        r_tos = 1..100;
        i_tos = 1..100;
```

Interpreters

- Basic Interpreter written in Pascal

```
procedure assignint (addr1 : integer; value : integer);
      begin
            ivar [addr1] := value;
      end;
procedure add( sym1, sym2 : symentry)
      begin
            ….
            if( sym1.type = 'real' and sym2.type = 'integer') then
                  addrealint (sym1.address, sym2.address);
            ….
      end
```

Interpreters

- Basic Interpreter written in Pascal

```
procedure addrealint (addr1, addr2 : integer);
    begin
        rvar[r_tos] := rvar[addr1] + ivar[addr2];
    end
begin{Main program}
    r_tos := 100;
    i_tos := 100;
    {Analyze a statement and call appropriate procedure}
end
```

Interpreters

- Example
  - Consider the Basic program

    real a, b

    integer c

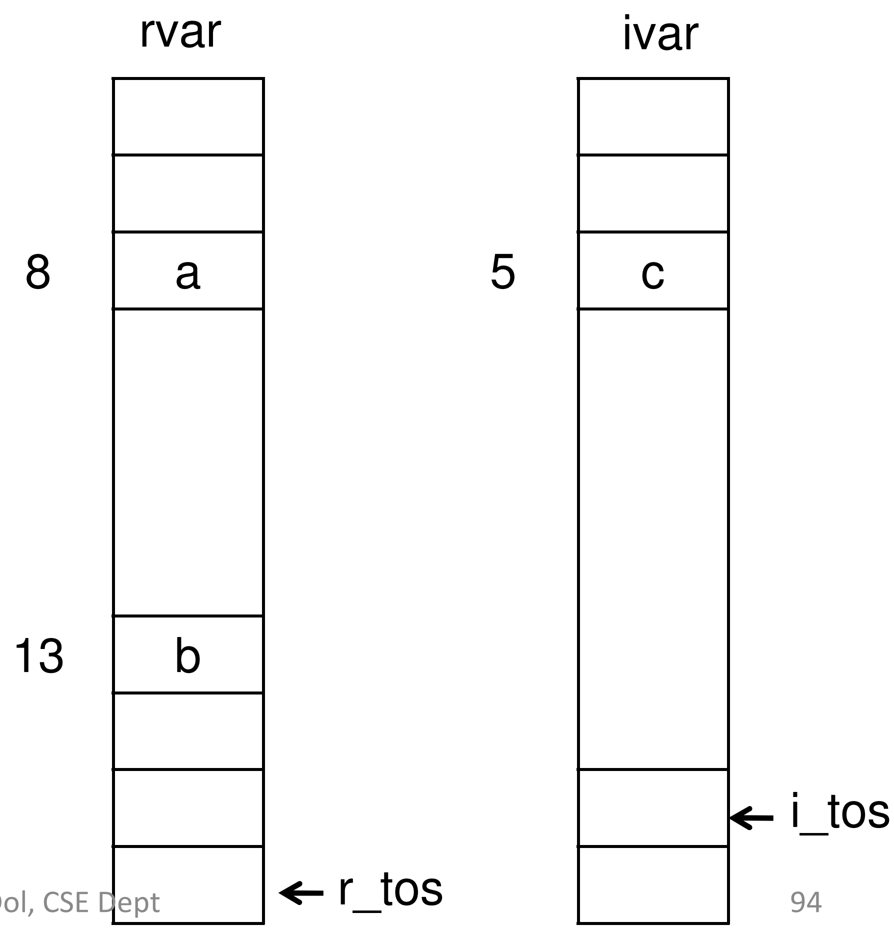
    let c = 7

    let b = 1.2

    a = b + c

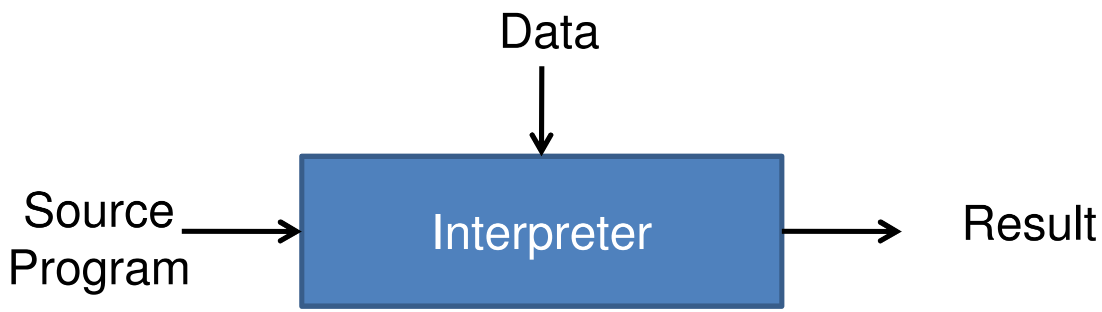
Symbol	Type	Address
a	real	8
b	real	13
c	int	5

Symbol Table

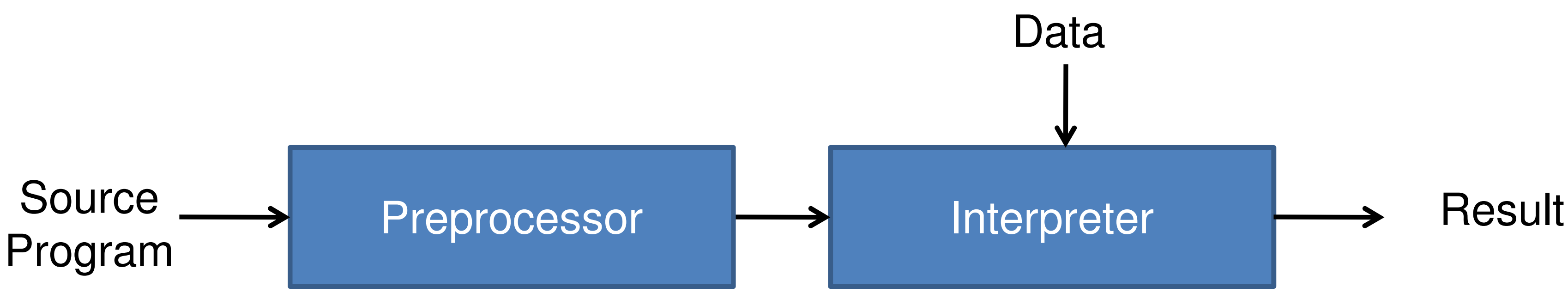


Interpreters

- Pure and Impure Interpreters



Pure Interpreter



Impure Interpreter

Interpreters

- Pure and Impure Interpreters
  - Impure Interpreter
    - It performs some preliminary processing of the source program to reduce analysis overhead during interpretation.
    - The preprocessor converts the program to an intermediate representation (i.e. intermediate code IC).
    - IC can be analyzed more efficiently than the source form of the program
    - E.g. intermediate code for a + b * c using postfix notation look like the following

S#17	S#4	S#29	*	+