Menoufia University

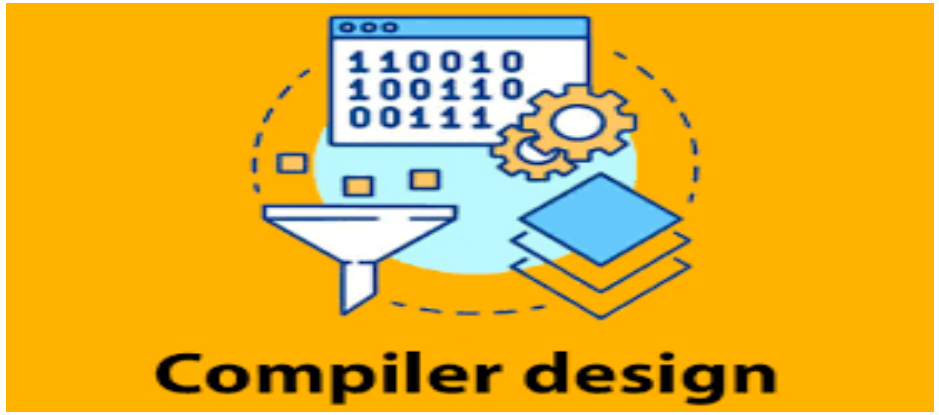
Faculty of computers & Information

Computer Science Department.

Compiler Design

4 Year – first Semester

Lecture 9



DR. Eman Meslhy Mohamed

Lecturer at Computer Science department

2023-2024

Code generation phase

- This is the phase which accepts as input the **syntax trees or stream of atoms** and produces, as output, the object language program in **binary coded instructions** in the proper format.
- The primary objective of the code generator is to convert atoms or syntax trees to instructions.
- It is also necessary to handle **register allocation** for machines that have several general purpose CPU registers.

Code generation phase

- Many designers view the construction of compilers as made up of two logical parts:-
  - **the front end**
  - **the back end**

- The front end consists of lexical and syntax analysis, it is machine-independent.

- The back end consists of code generation and optimization, it is machine- dependent.

The front and back end

- This separation into front and back ends simplifies things in two ways when constructing compilers for new machines or new languages.
- First, if we are implementing a compiler for a new machine, and we already have compilers for our old machine, all we need to do is write the back end, since the front end is not machine dependent.
- For example, if we have a java compiler for an IBM and we wish to implement java on a Macintosh machine.
- We can use the front end of the existing java compiler. This means that we need to write only the back end of the new compiler

The front and back end

- Second, if we are implementing a compiler for a new programming language on an existing computer. In this case, we can make use of the back end already written for our existing compiler.
- All we need to do is rewrite the front end for the new language.
- For example, suppose we have a Pascal compiler for the Macintosh, and we wish to construct an Ada compiler for the Macintosh.
- All we need to do is rewrite the front end for the Ada language and use the back end of the Mac machine .

Converting Atoms to Instructions

- Each atom class would result in a different instruction or sequence of instructions.
- ADD atom is translated into:
  - a LOD (Load Into Register) instruction,
  - followed by an ADD instruction,
  - followed by a STO (Store Register To Memory) instruction

- Example of a translation of an ADD atom

```
(ADD, A, B, T1)    →        LOD    R1,A
                            ADD    R1,B
                            STO    R1,T1
```

Converting Atoms to Instructions

- Conditional Branch atoms (called TST atoms) would normally be implemented as:
  - a Load, Compare, and Branch, depending on the architecture of the target machine.

- The MOV (move data from one memory location to another) atom could be implemented as a Load followed by a Store.

Example

- Example : the java statement if (a>b) a = b * c; might result in the following sequence of atoms:
- Translate these atoms to instructions for a Load/Store architecture.

```
(TST, A, B, 4, L1)
(MUL, B, C, A)
(LBL L1)
```

```
LOD r1,a // Load a into reg. r1
CMP r1,b,4 // Compare a <= b?
JMP L1 // Branch if true
LOD r1,b
MUL r1,c // r1 = b * c
STO r1,a // a = b * c
L1:
```

Single Pass & Multiple Passes

- There are several different ways of approaching the design of the code generation phase.
- Single pass code generator is a code generator which scans the file of atoms once.
- Multiple pass code generator is a code generator which scans the file of atoms more than once.

Single Pass & Multiple Passes

- The most significant problem relevant to deciding whether to use a single or multiple pass code generator has to do with forward jumps.

- If a Jump atom is executed with a destination that is a higher memory address than the Jump instruction (the label to which it is jumping has not yet been encountered)

- It will not be possible to generate the Jump instruction completely at this time.

Example

Atom	Location	Instruction
(ADD, A, B, T1)	4	LOD R1,A
	5	ADD R1,B
	6	STO R1,T1
(JMP,L1)	7	CMP 0,0,0
	8	JMP ?
(LBL, L1)		(L1 = 9)

Single pass compilers Solution

- **Single pass compilers** resolve it by keeping a table of Jump instructions which have forward Jump.
- Each Jump instruction with a forward reference is entered into a **fixup table**, along with the Label to which it is jumping.
- As each **Label definition is encountered**, it is entered into a **table of Labels**, along with its address value.
- When all of the atoms have been read, the code generator can **revisit** all of the Jump instructions in the **Fixup table** and **fill** in their destination addresses.

Single pass compilers Solution

Atom	Loc	Instruction		Fixup Table Loc	Fixup Table Label	Label Table Label	Label Table Value
(ADD,A,B,T1)	4	LOD	R1,A				
	5	ADD	R1,B				
	6	STO	R1,T1				
(JMP,L1)	7	CMP	0,0,0				
	8	JMP	0	8	L1		
(LBL,L1)						L1	9
...							
EOF							
	8	JMP	9				

Multiple pass code generators

- Multiple pass code generators do not require a Fixup table.
- In this case, the first pass of the code generator does nothing but build the table of Labels, storing a memory address for each Label.
- Then, in the second pass, all the Labels will have been defined, and each time a Jump is encountered its destination Label will be in the table, with an assigned memory address.

Multiple pass code generators

Begin First Pass:

Atom	Loc	Instruction	Label Table Label	Value
(ADD,a,b,T1)	4-6			
(JMP,L1)	7-8			
(LBL,L1)			L1	9
...				
EOF				

Begin Second Pass:

Atom	Loc	Instruction
(ADD,a,b,T1)	4	LOD r1,a
	5	ADD r1,b
	6	STO r1,T1
(JMP,L1)	7	CMP 0,0,0
	8	JMP 9
(LBL,L1)		
...		
EOF		

- The following atom string resulted from the java statement

  while (i<=x) { x = x+2; i = i*3; }.

- Translate it into instructions as in (1) a single pass code generator using a Fixup table and (2) a multiple pass code generator.

```
(LBL, L1)
(TST, i, x, 3, L2)          // Branch to L2 if i>x
(ADD, x, 2, T1)
(MOV, T1, , x)
(MUL, i, 3, T2)
(MOV, T2, , i)
(JMP, L1)                   // Repeat the loop
(LBL, L2)                   // End of loop
```

Single pass code generators

(1) Single Pass

```
                                                  Fixup Table        Label Table
Atom              Loc Instruction                 Loc     Label      Label   Value
(LBL, L1)         0                                                  L1      0
(TST,i,x,,3,L2)   0   CMP  i,x,3
                  1   JMP  ?                     1       L2
(ADD, X, 2, T1)   2   LOD  R1,x
                  3   ADD  R1,2
                  4   STO  R1,T1
(MOV, T1,, x)     5   LOD  R1,T1
                  6   STO  R1,x
(MUL, i, 3, T2)   7   LOD  R1,i
                  8   MUL  R1,3
                  9   STO  R1,T2
(MOV, T2,, i)     10  LOD  R1,T2
                  11  STO  R1,i
(JMP, L1)         12  CMP  0,0,0
                  13  JMP  0
(LBL, L2)         14                                               L2      14

...
                  1   JMP  14
```

Multiple pass code generators

(2) Multiple passes

Begin First Pass:

Atom	Loc	Instruction
(LBL, L1)	0	
(TST,i,x,,3,L2)	0	
(ADD, X, 2, T1)	2	
(MOV, T1,, x)	5	
(MUL, i, 3, T2)	7	
(MOV, T2,, i)	10	
(JMP, L1)	12	
(LBL, L2)	14	

...

Label Table

Label	Value
L1	0
L2	14

Begin Second Pass:

Atom	Loc	Instruction
(LBL, L1)	0	
(TST,i,x,,3,L2)	0	CMP i,x,3
	1	JMP 14
(ADD, X, 2, T1)	2	LOD R1,x
	3	ADD R1,2
	4	STO R1,T1
(MOV, T1,, x)	5	LOD R1,T1
	6	STO R1,x
(MUL, i, 3, T2)	7	LOD R1,i
	8	MUL R1,3
	9	STO R1,T2
(MOV, T2,, i)	10	LOD R1,T2
	11	STO R1,i
(JMP, L1)	12	CMP 0,0,0
	13	JMP 0
(LBL, L2)	14	

Register Allocation

- Some computers are designed with a single arithmetic register, called an accumulator, in which all arithmetic operations are performed.
- Other computers (such as the Intel 8086) have only a few CPU registers, and they are not general purpose registers;
- However, most modern architectures have many CPU registers;
- The main objective in register allocation is to maximize utilization of the CPU registers, and to minimize references to memory locations.

Register Allocation

- An example, showing the importance of smart register allocation, for the two statement program segment:

A = B + C *D ;

B = A - C *D ;

Simple Register Allocation

```
LOD     R1,C
MUL     R1,D
STO     R1,Temp1
LOD     R1,B
ADD     R1,Temp1
STO     R1,A
LOD     R1,C
MUL     R1,D
STO     R1,Temp2
LOD     R1,A
SUB     R1,Temp2
STO     R1,B
```

Smart Register Allocation

```
LOD     R1,C
MUL     R1,D        C*D
LOD     R2,B
ADD     R2,R1       B+C*D
STO     R2,A
SUB     R2,R1       A-C*D
STO     R2,B
```

Register Allocation

- Here, we will discuss an algorithm which determines how many registers will be needed to evaluate an expression without storing sub-expressions to temporary memory locations.

- This register allocation algorithm will require **a syntax tree** for an expression to be evaluated.

- Each node of the syntax tree will have a weight associated with it which tells us how many registers will be needed to evaluate each sub-expression **without storing to temporary** memory locations.

Register Allocation

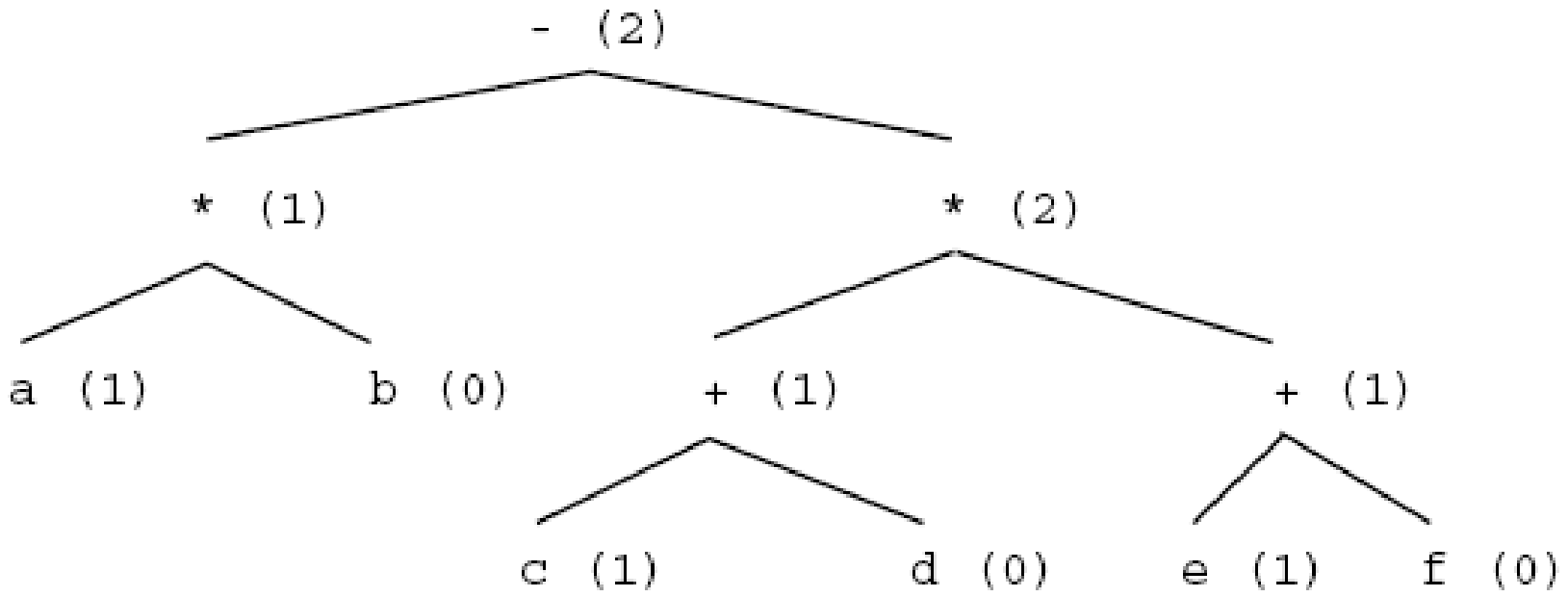
- **Each leaf node which is a left** operand will have a weight of one, and **each leaf node which is a right** operand will have a weight of zero.

- The weight of each interior node will be computed from the weights of its two children as follows:

  - If the two children have different weights, the parent's weight is the **maximum** of the two children.
  -
  - If the two children have the same weight, **w**, then the parent's weight is **w+1**.

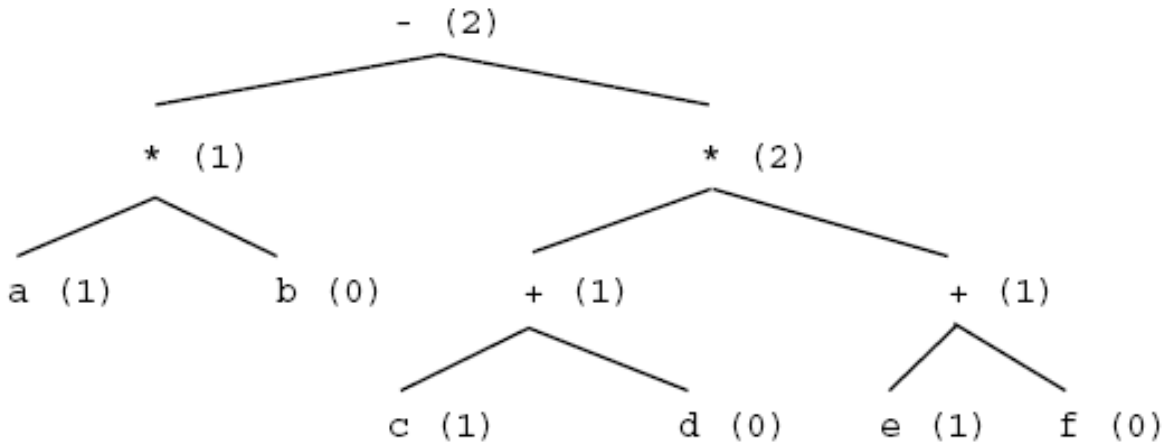
Example

- The weighted syntax tree for the expression a*b -(c+d) * (e+f) should require two registers.



Register Allocation

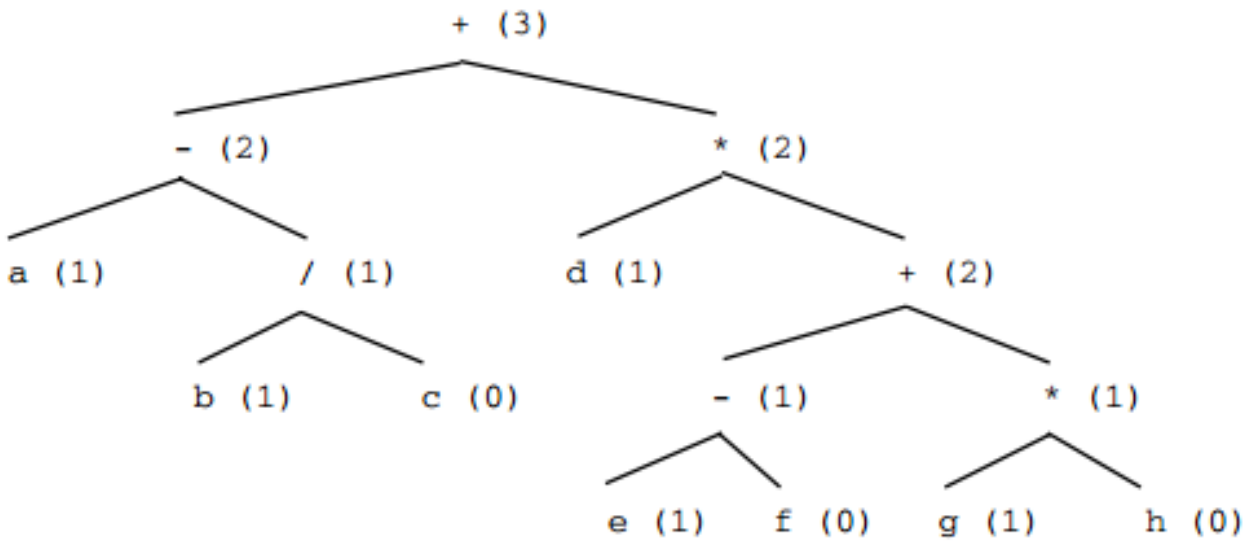
- How can we generate code for this expression.
- We do this by evaluating the operand having greater weight, first.
- If both operands of an operation have the same weight, we evaluate the left operand first.




```
LOD   R1,c
ADD   R1,d        R1 = c + d
LOD   R2,e
ADD   R2,f        R2 = e + f
MUL   R1,R2       R1 = (c+d) * (e+f)
LOD   R2,a
MUL   R2,b        R2 = a * b
SUB   R2,R1       R2 = a*b - (c+d)*(e+f
```

Example

- Use the register allocation algorithm to show a weighted syntax tree for the expression a - b/c + d * (e-f + g*h)




```
LOD     r1,a
LOD     r2,b
DIV     r2,c        b/c
SUB     r1,r2       a - b/c
LOD     r2,e
SUB     r2,f        e - f
LOD     r3,g
MUL     r3,h        g * h
ADD     r2,r3       e - f + g * h
LOD     r3,d
MUL     r3,r2       d * (e-f + g*h)
ADD     r1,r3       a - b/c + d * (e-f + g*h)
```

Thanks