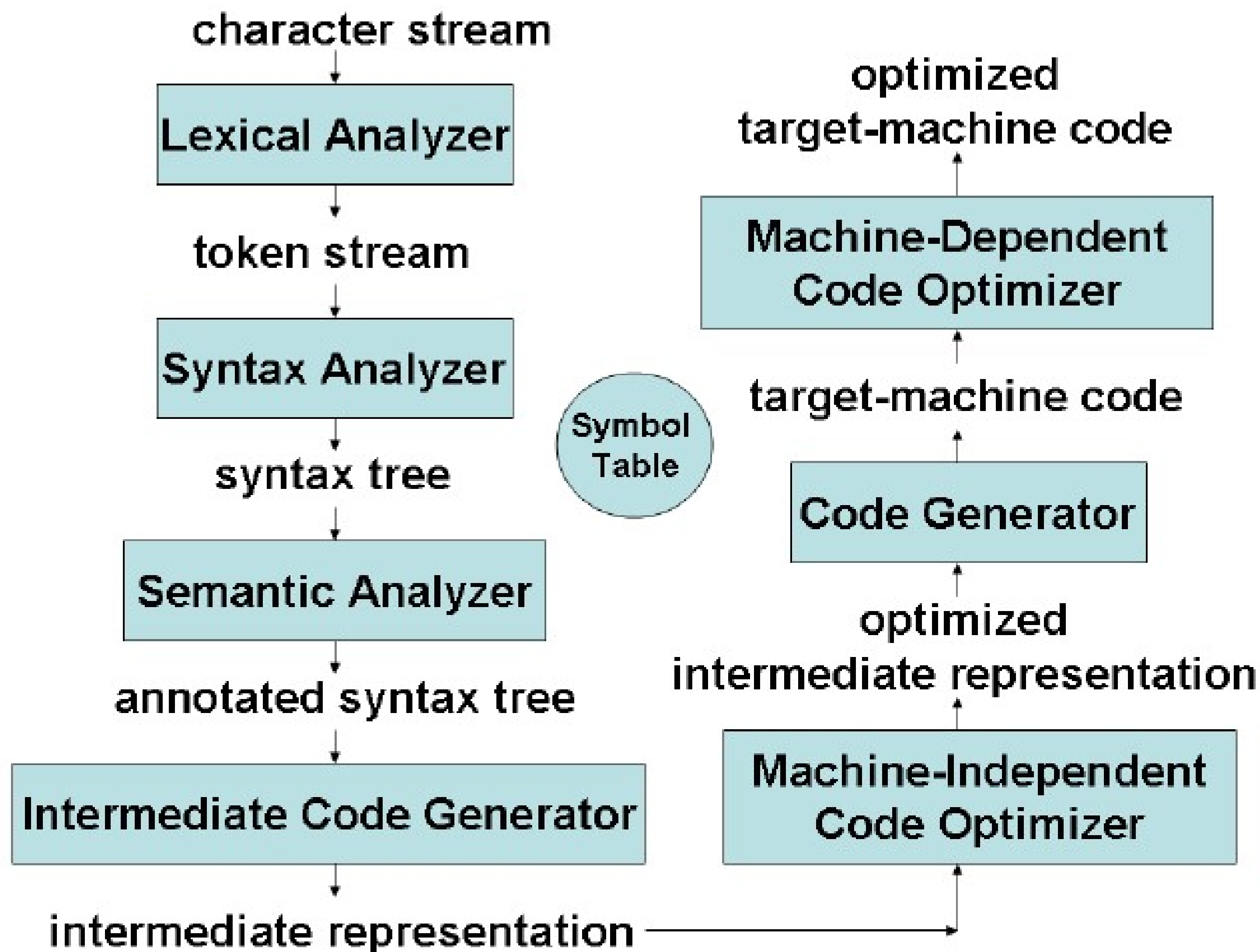
Chapter # 11
Inermediate Code Generation

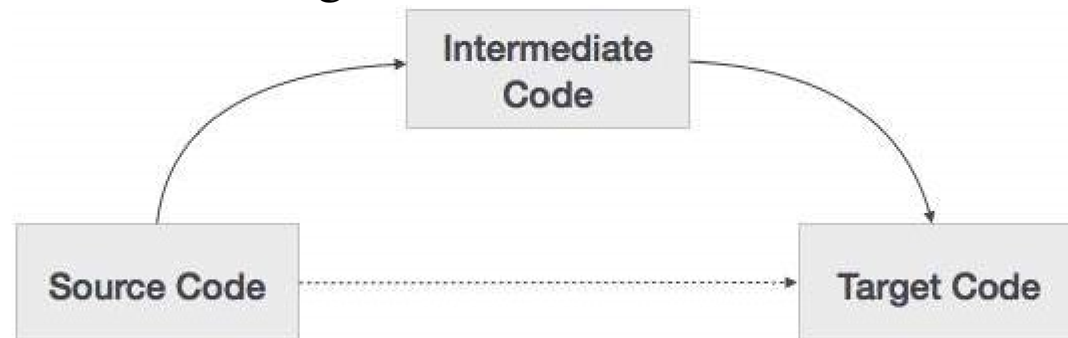
Dr. Shaukat Ali
Department of Computer Science
University of Peshawar

Compiler Overview



Introduction

- Intermediate code is the interface between front-end and back-end in a compiler
- Ideally the details of source language are confined to the front-end and the details of target machines to the back-end

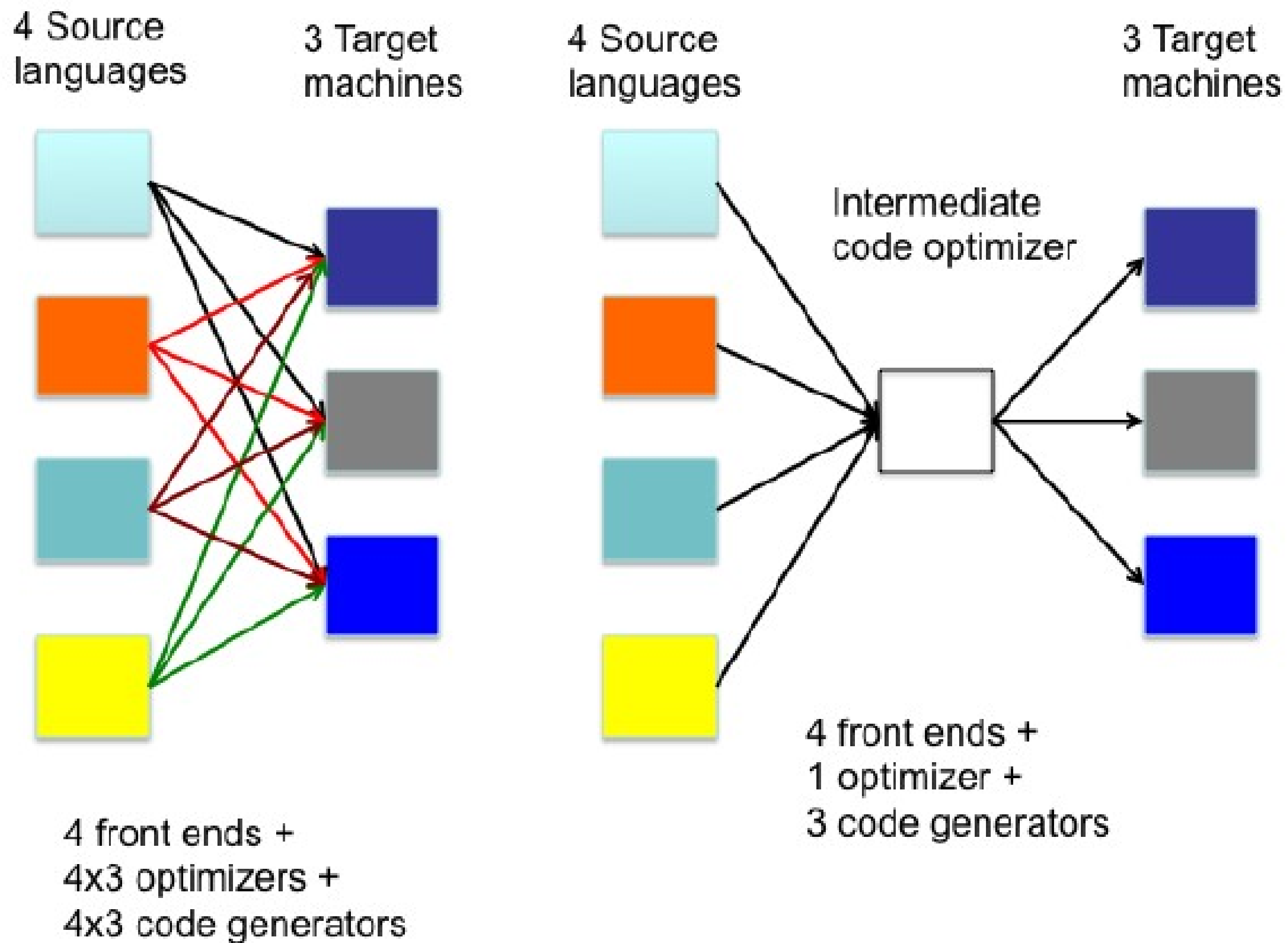


- A source code can directly be translated into its target machine code
  - Why at all we need to translate the source code into an intermediate code which is then translated to its target code?

Why Intermediate Code?

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers
  - The second part of compiler, synthesis, is changed according to the target machine
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code

Why Intermediate Code?



Why Intermediate Code?

- While generating machine code directly from source code is possible, it entails problems
  - With m languages and n target machines, we need to write m front ends, m x n optimizers, and m x n code generators
  - The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
  - This means just m front ends, n code generators and 1 optimizer

Intermediate Representation

- Intermediate codes can be represented in a variety of ways and they have their own benefits
  - **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred
  - **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations
- Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Intermediate Code Generation

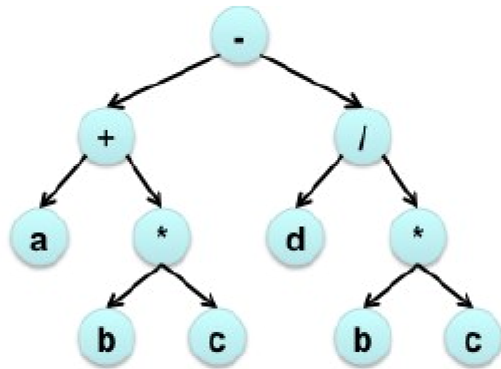
- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)
- Intermediate code is represented in three-address space but the type of intermediate code implementation is based on the compiler designer
  - Quadruples, triples, indirect triples are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations

Three-Address Code

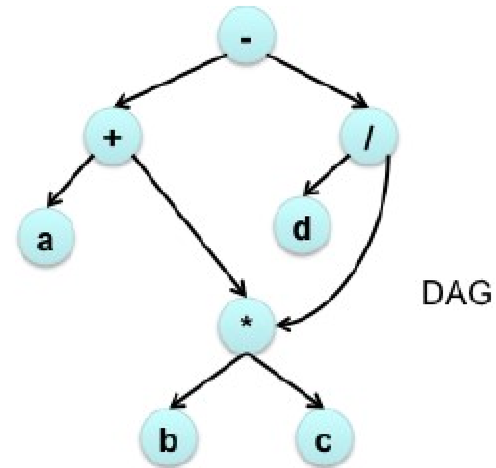
- Instructions are very simple : LHS is the target and the RHS has at most two sources and one operator
- RHS sources can be either variables or constants
- Examples: `a = b + c, x = -y, if a > b goto L1`
- Three-address code is a generic form and can be implemented as quadruples, triples, indirect triples
- Example: The three-address code for `(a+b*c)-(d/(b*c))` is below

IR Code is Made From

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



Syntax tree



DAG

- Intermediate Code

1. t1 = b*c
2. t2 = a+t1
3. t3 = b*c
4. t4 = d/t3
5. t5 = t2-t4

$t1 = b * c$

$t2 = a + t1$

$t3 = d / t1$

$t4 = t2 - t3$

Example

- The intermediate code produced from DAG is more compact as compared AST
- a = b * (minus c) + b * (minus c)

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

(a) Code for the syntax tree.

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

(b) Code for the dag.

Instructions in 3 – Address Space (1)

- **Assignment instructions:** `a = b biop c`, `a = uop b`, and `a = b` (copy)
  - Where
    - `biop` is any binary arithmetic, logical, or relational operator
    - `uop` is any unary arithmetic (++, --, conversion) or logical operator (!)
    - Conversion operators are useful for converting integers to floating point numbers, etc.
- **Jump instructions:**

  `goto L` (unconditional jump to L),

  `if t goto L` (it t is true then jump to L),

  `if a relop b goto L` (jump to L if a relop b is true),
  - where
    - `L` is the label of the next three-address instruction to be executed
    - `t` is a boolean variable
    - `a` and `b` are either variables or constants

Instructions in 3 – Address Space (2)

- Functions:

  `func begin <name>` (beginning of the function),

  `func end` (end of a function),

  `param p` (place a value parameter p on stack),

  `refparam p` (place a reference parameter p on stack),

  `call f, n` (call a function f with n parameters),

  `return` (return from a function),

  `return a` (return from a function with a value a)

- Indexed copy instructions:

  `a = b[i]` (a is set to contents(baseaddress(b)+offset(i)), where b is (usually) the base address of an array

  `a[i] = b` (ith location of array a is set to b)

Instructions in 3 – Address Space (3)

- Pointer assignments:

  `a = &b` (a is set to the address of b, i.e., a points to b)

  `*a = b` (contents(contents(a)) is set to contents(b))

  `a = *b` (a is set to contents(contents(b)))

Intermediate Code – Example 1

C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

```
    dot_prod = 0;           |    T6 = T4[T5]
    i = 0;                  |    T7 = T3*T6
L1: if(i >= 10)goto L2      |    T8 = dot_prod+T7
    T1 = addr(a)            |    dot_prod = T8
    T2 = i*4                |    T9 = i+1
    T3 = T1[T2]             |    i = T9
    T4 = addr(b)            |    goto L1
    T5 = i*4                |L2:
```

Intermediate Code – Example 2

C-Program

```
int a[10], b[10], dot_prod, i; int* a1; int* b1;
dot_prod = 0; a1 = a; b1 = b;
for (i=0; i<10; i++) dot_prod += *a1++ * *b1++;
```

Intermediate code

```
    dot_prod = 0;           |    b1 = T6
    a1 = &a                 |    T7 = T3*T5
    b1 = &b                 |    T8 = dot_prod+T7
    i = 0                   |    dot_prod = T8
L1: if(i>=10)goto L2        |    T9 = i+1
    T3 = *a1                |    i = T9
    T4 = a1+1               |    goto L1
    a1 = T4                 |L2:
    T5 = *b1
    T6 = b1+1
```

Intermediate Code – Example 3

C-Program (function)

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
```

Intermediate code

```
    func begin dot_prod |     T6 = T4[T5]
    d = 0;              |     T7 = T3*T6
    i = 0;              |     T8 = d+T7
L1: if(i >= 10)goto L2  |     d = T8
    T1 = addr(x)        |     T9 = i+1
    T2 = i*4            |     i = T9
    T3 = T1[T2]         |     goto L1
    T4 = addr(y)        |L2: return d
    T5 = i*4            |     func end
```

Intermediate Code – Example 4

C-Program (main)

```
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Intermediate code

```
    func begin main
    refparam a
    refparam b
    refparam result
    call dot_prod, 3
    p = result
    func end
```

Intermediate Code – Example 5

C-Program (function)

```
int fact(int n){
   if (n==0) return 1;
   else return (n*fact(n-1));
}
```

Intermediate code

```
func begin fact       |       T3 = n*result
if (n==0) goto L1     |       return T3
T1 = n-1              |   L1: return 1
param T1              |       func end
refparam result       |
call fact, 2          |
```

Data Structures for 3 – Address Code

- Quadruples
  - Has four fields: op, arg1, arg2 and result
  - Temporaries are used
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Quadruples

- In quadruples representation, there are four fields for each instruction: op, arg1, arg2, result
  - Binary ops have the obvious representation
  - Unary ops do not use arg2
  - Operators like param does not use either arg2 result
  - Jumps put the target label into the result
- The quadruples implement the three address space for the expression

```
a = b * (-c) + b * (-c)
```

Quadruples

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

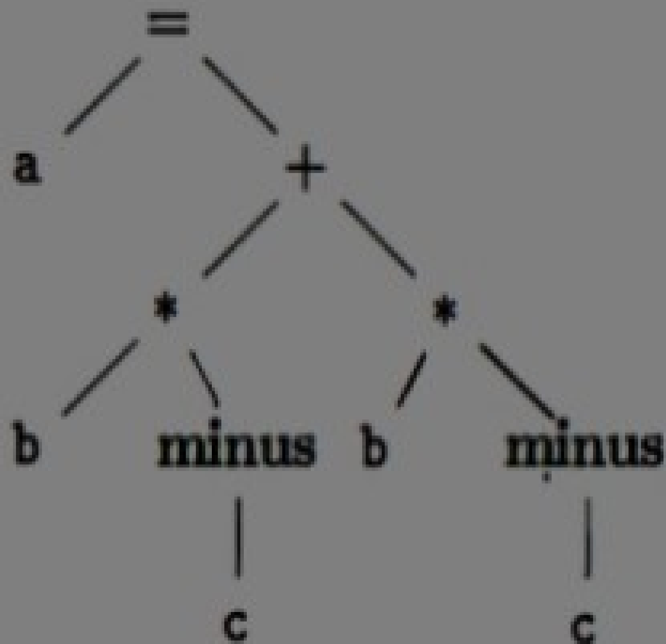
(b) Quadruples

Archana, Govt Engg College, Trivandrum

Triples

- Triples has only three fields for each instruction: op, arg1, arg2
- The result of an operation x op y is referred by its position
- Triples are equivlant to signature of nodes in DAG or syntax tree
- Triples and DAG are equivalent representations only for expressions
- Ternary operation like X[i] =Y requires two entries in the triple structure; similarly for Y = X[i]

Triples



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

ya Arunan, Govt. Engg College, Trivandrum

Indirect Triples

- These consist of a listing of pointers to triples; rather than a listing of the triples themselves
- The triples consists of three fields: op, arg1, arg2
- The arg1 or arg2 could be pointers

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Example

- a = b * (minus c) + b * (minus c)

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

Quadruples

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

Triples

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

Indirect Triples

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

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

Example 2

3-address code

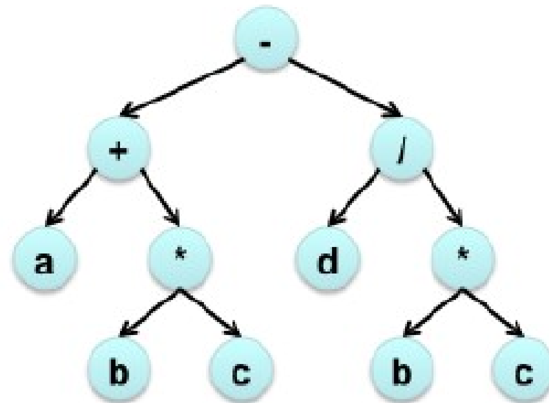
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

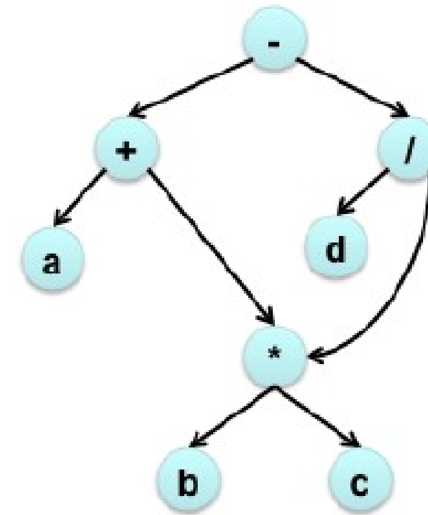
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)



Syntax tree



DAG

- End of Chapter # 11