Intermediate Code Generation
Module 4

Prepared by

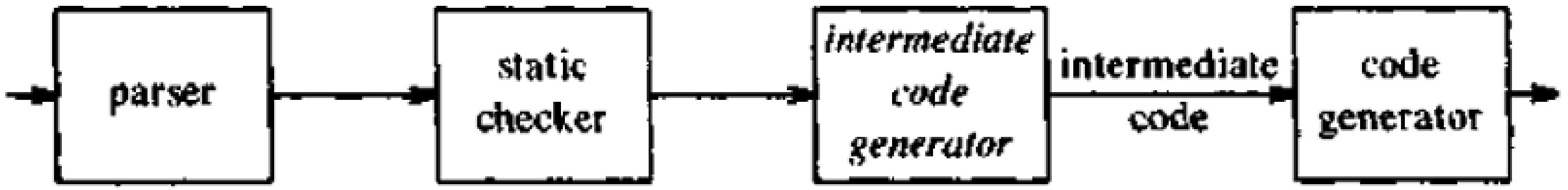
Jesna J S

Assistant Professor CSE, TKMCE

Intermediate Code Generation

- In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code.
- Details of the target language are confined to the back end, as far as possible.
- Benefits of using a machine-independent intermediate form are:
    1. Retargeting is facilitated: A compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
    2. A machine independent code optimizer can be applied to the intermediate representation.

- The intermediate code generator is crucial because it separates machine-independent optimizations from machine-dependent code generation.



- *The parser takes the source code and checks its syntax based on the grammar of the programming language.*
- *It converts the source code into a parse tree or an* ***abstract syntax tree (AST)*** *for further processing.*
- *The static checker phase performs semantic analysis, ensuring that the program follows the language's rules. It checks for type mismatches, undeclared variables, and scope violations. If errors are found, they are reported to the user.*
- *The intermediate code generator component converts the verified AST into an intermediate representation (IR).*

Intermediate Languages

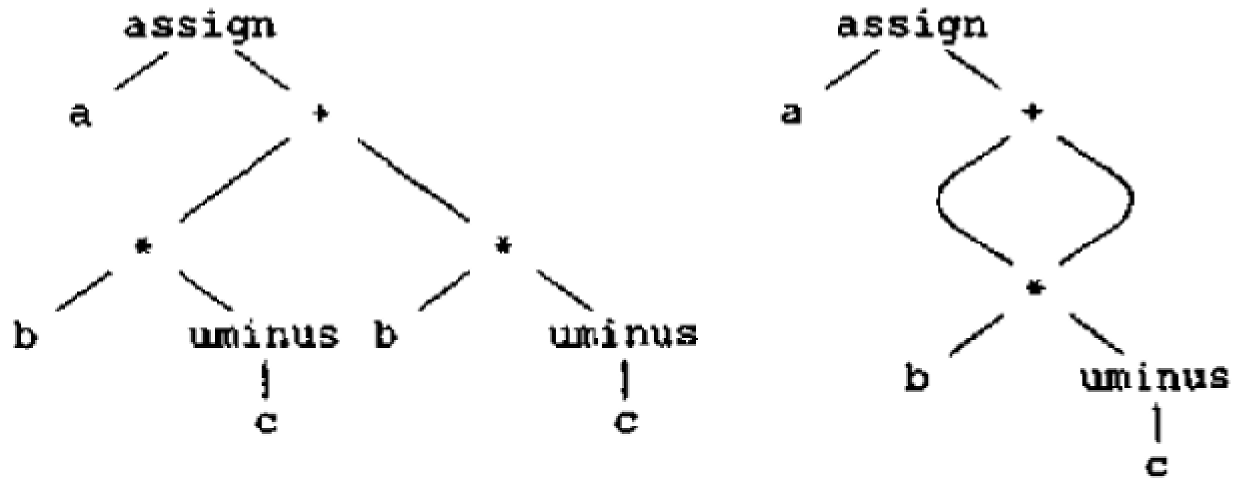
1. Syntax Tree
2. DAG (Directed Acyclic Graph)
3. Postfix Notation
4. 3 Address code

1. Abstract Syntax Tree

- A syntax tree, also called a parse tree, represents the complete structure of a program based on grammar rules. It contains all elements of the grammar, including operators, keywords, and parentheses. The structure is directly derived from the context-free grammar (CFG) of the language.
- An AST (Abstract Syntax Tree) is a simplified, more compact version of the syntax tree. It removes unnecessary details like parentheses and grammar-specific rules. It focuses only on the meaningful structure of the program.

2. DAG (Directed Acyclic Graph)

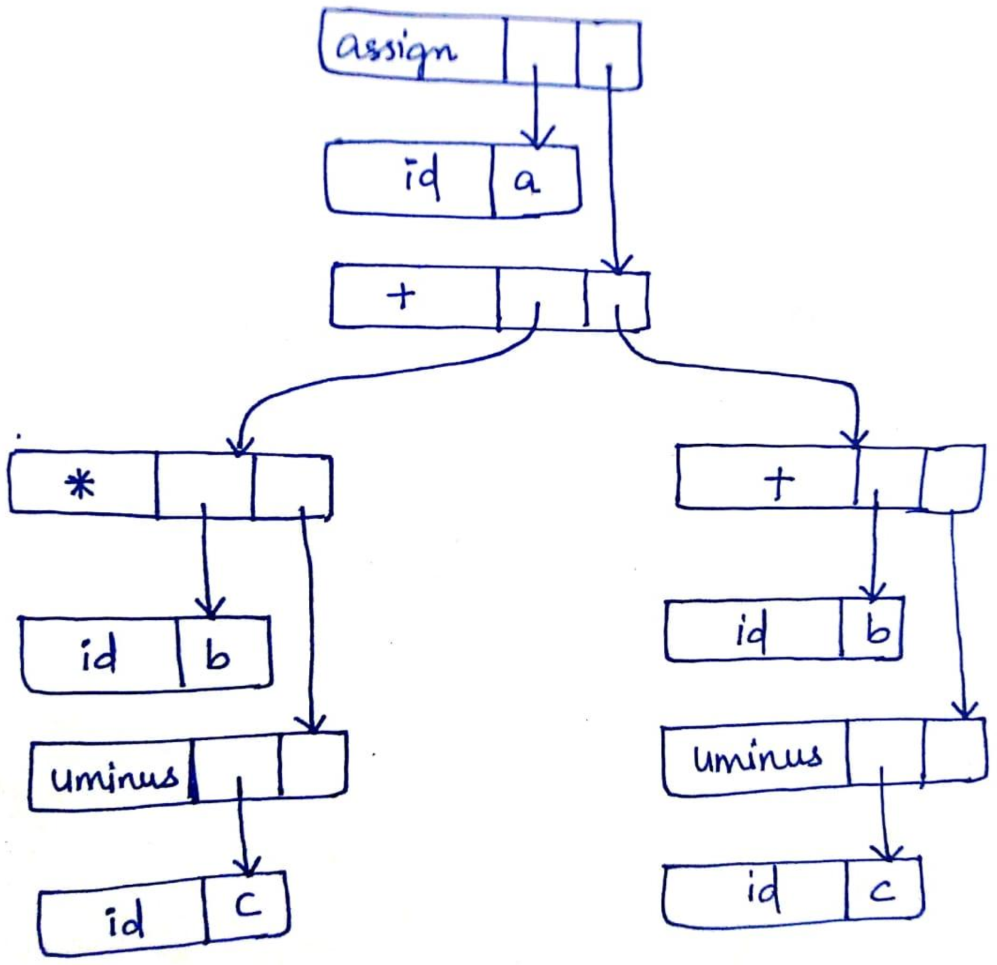
Syntax tree and DAG for the assignment a= b*-c + b*-c

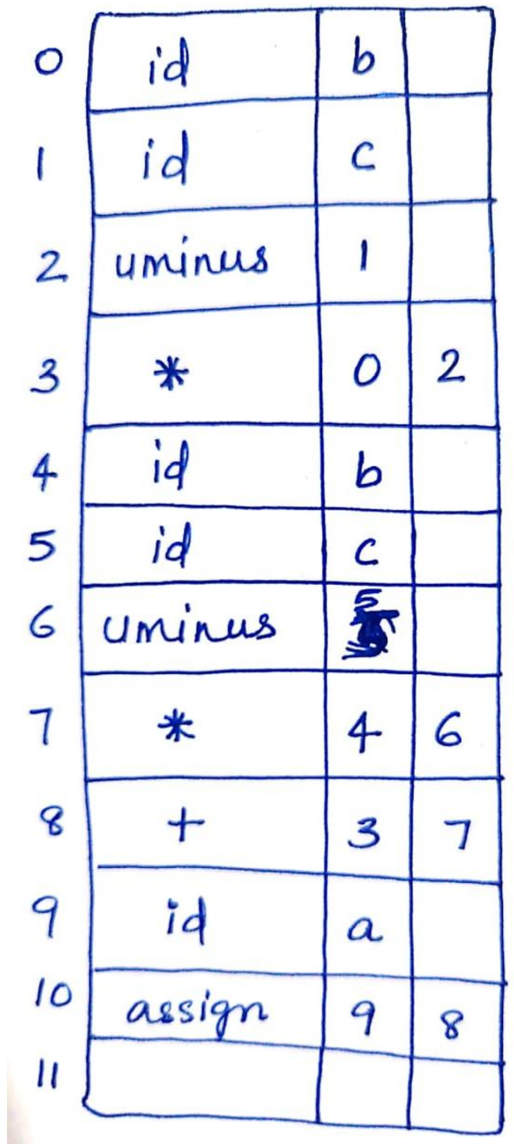


(a) Syntax tree. (b) Dag.

Fig. 8.2. Graphical representations of `a := b* - c + b* - c`.

Syntax tree representations of a=b-c + b*-c*





0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11			

- *In the first representation each node represented as a record with a field for its operator and additional fields for pointers to its children.*
- *In the second representation, nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.*

Postfix Notation

- **Postfix notation**, also known as **Reverse Polish Notation (RPN)**, is a mathematical notation where **operators are placed after operands** instead of between them.
- It is commonly used in **intermediate code generation** in compilers because it eliminates the need for parentheses and follows a **stack-based evaluation** approach.
- Linearized representation of syntax tree
- Operators can be evaluated in the order in which they appear in the string

- Example:

  Source String : a := b * -c + b * -c

  Postfix String: a b c uminus * b c uminus * + assign

-

Three-Address Code

- **Three-Address Code (TAC)** is a widely used **intermediate representation (IR)** in compilers.
- It represents statements using at most **three operands** and a single operator in each instruction.

Advantages of Three-Address Code representation

- Simplifies optimization: Easier for compilers to perform optimizations like constant folding and dead code elimination.
- Machine-independent: Provides an abstract representation before generating machine-specific code.
- Easier to generate assembly code: The structure closely resembles low-level assembly instructions

- General form of 3 address code:

  x = y op z

Where x, y, and z are operands (variables or constants) ,op is an operator (arithmetic, relational, logical, etc.)

- If only one operand is needed, the format is: x = op y
- *Example*: Q. **(A + B) * (C - D)**

  t1 = A + B

  t2 = C - D

  t3 = t1 * t2

  Q. **a = b*-c + b*-c**

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

Example: Intermediate Code representation of statement a=b*-c + b*-c

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

code for syntax tree

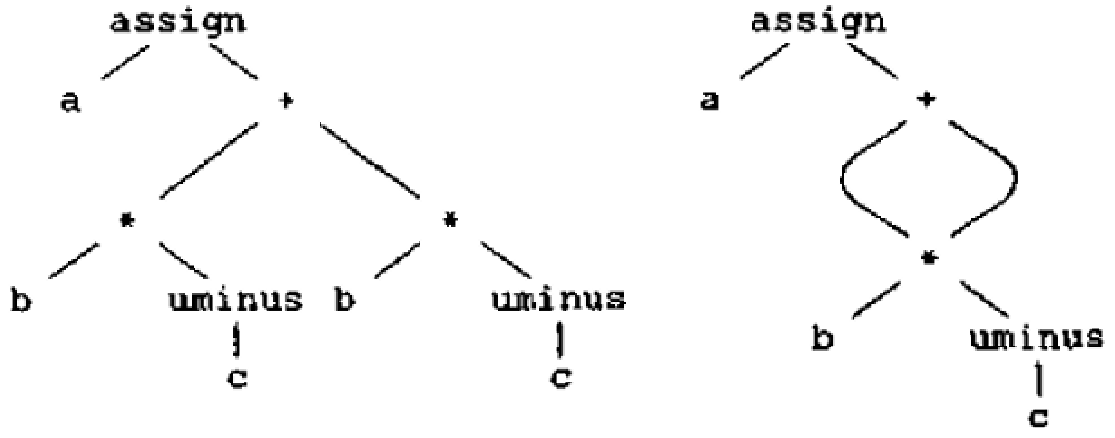


Fig. 8.2. Graphical representations of a := b* - c + b* - c.

$t_1 := -c$

$t_2 := b * t_1$

$t_5 := t_2 + t_2$

$a = t_5$

code for dag

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

t1=b*c

t2=a+t1

t3=b*c

t4=d/t3

t5=t2-t4

Conditional Statements and Looping Statements

```
while(i<10)
{
        x=0;
        i=i+1;
}
```

For converting to 3AC, looping statements must be converted to if statements and goto statements.

<u>3AC</u>

```
L1: if i<10 goto L2
     goto Lnext
L2: x=0;
    t1=i+1;
    i=t1;
    goto L1
Lnext:
```

- ***Source***:

```
if ( x + y*z > x*y + z)
   a = 0;
```

- ***Three Address Code***:

```
  tmp1 = y*z
  tmp2 = x+tmp1           // x + y*z
  tmp3 = x*y
  tmp4 = tmp3+z           // x*y + z
  if (tmp2 > tmp4) goto L
  goto L1
L:a = 0
L1:
```

```
while (A < C and B > D)
{
  if A = 1
        C = C + 1
  else
        while A <= D
                A = A + B
}
```

```
1. if (A < C) goto (3)
2. goto (15)
3. if (B > D) goto (5)
4. goto (15)
5. if (A = 1) goto (7)
6. goto (10)
7. T1 = c + 1
8. c = T1
9. goto (1)
10. if (A <= D) goto (12)
11. goto (1)
12. T2 = A + B
13. A = T2
14. goto (10)
15.
```

```
while(a<b)
{
            if((c<d) &&(e>f))
                 x=y+z;
            else
                x=y-z;
}
```

In this example, if statement evaluates two expressions

First convert the expression to 3AC and write it as follows.

3AC

```
L1: if a<b goto L2
      goto Lnext
L2: t1=c<d;
      t2=e>f;
      if (t1&&t2) goto L3
      goto L4
L3: t3=y+z;
      x=t3
     goto L1
L4:t4=y-z;
      x=t4
     goto L1
Lnext:
```

Practice Questions

1. do
```
    {
      x=y+z-a;
    }while(a<b);
```

2. while(p<q and r>s)
```
  {
    if(p==1)
      q=q+1;
    else
      while(p<=s)
              p=a+3;
  }
```

3. sum=0;
```
 for(j=1;j<=10;j++)
sum=sum+a[j]+b[j];
```

4. switch(i+j)
```
 {
   case 1: x=y+z;
   case 2: p=q+r;
   default: u=u+v;
 }
```

5. switch(i+j)
```
 {
   case 1: x=y+z; break;
   case 2: p=q+r; break;
   default: u=u+v;
 }
```

Types of Three-Address Statements

1. Assignment statements
   - x := y op z, where op is a binary arithmetic or logical operation.
2. Assignment instructions
   - x : = op y, where op is a unary operation .
   - Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that for example, convert a fixed-point number to a floating-point number.
3. Copy statements
   - x : = y where the value of y is assigned to x.
4. Unconditional jump
   - goto L
   - The three-address statement with label L is the next to be executed

4. Conditional jump
   - if x relop y goto L
   - This instruction applies a relational operator (<, =, =, etc,) to x and y, and executes the statement with label L next if x stands in relation relop to y.
   - If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.
5. Indexed Assignments
   - Indexed assignments of the form **x = y[i] or x[i] = y**
6. Address and pointer assignments
   - Address and pointer operator of the form

     x := &y, **x := *y** and ***x := y**

7. Procedural call and return

param x and call p, n for procedure calls and return y, where y representing a returned value is optional.

Their typical use is as the sequence of three-address statements

```
param x1
param x2
……….
param xn
call p, n
```

The integer n indicating the number of actual-parameters in "call p

Implementation of Three-Address Statements

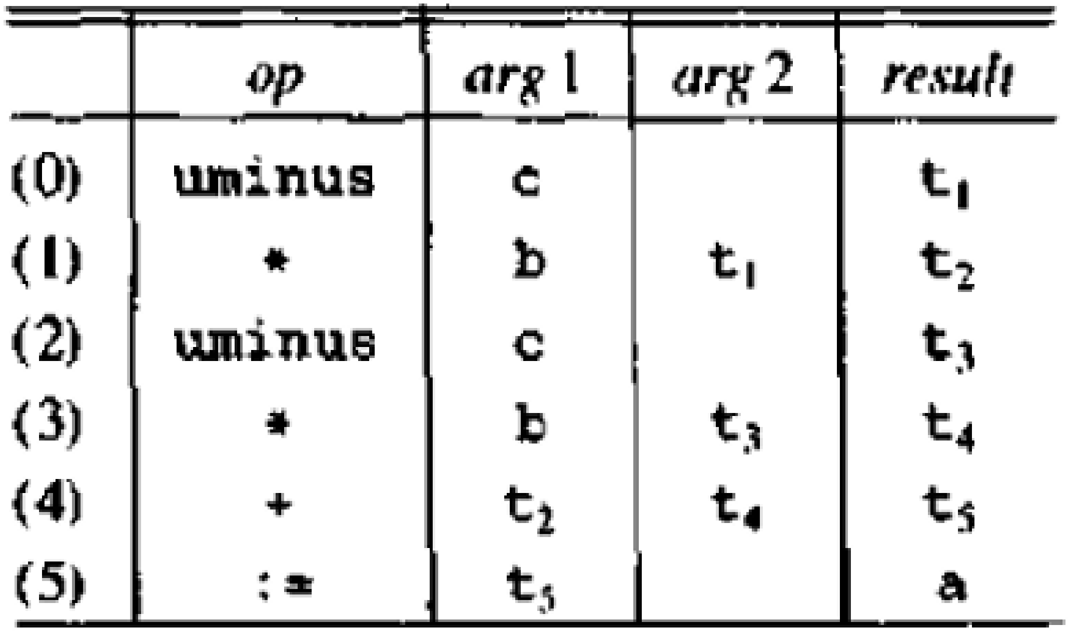
- A three-address statement is an abstract form of intermediate code.
- In a compiler, these statements can be implemented as records with fields for the operator and the operands.
- And such, representations are

1. **QUADRUPLES**
2. **TRIPLES**

QUADRUPLES

- A quadruple is a record structure with four fields, which are op, ag1, arg2 and result
- The op field contains an internal code for the operator.
- The three address statement x:= y op z is represented by placing y in arg1, z in arg2 and x in result.
- The contents of arg1, arg2, and result are normally **pointers to the symbol table entries** for the names represented by these fields.
- Hence temporary names must be entered into the symbol table as they are created.

Quadruples representation of statement **a=b*-c + b*-c**



	op	*arg* 1	*arg* 2	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

<u>Code for syntax tree</u>

Q. Reprasent the following statements in quadruple form

- **a + b * c / e ^ f + b * a**
- **p+q-r+s*t+j**

```
L1: if i<10 goto L2
     goto Lnext
L2: x=0;
    t1=i+1;
    i=t1;
    goto L1
Lnext:
```

	Op	Arg1	Arg2	Result
(0)	JLE	i	10	(2)
(1)	goto			(6)
(2)	=	0		x
(3)	+	i	1	t1
(4)	=	t1		i
(5)	goto			(0)
(6)				

- Look at the first statement-Jump Less Than(JLE) is the operator, i and 10 are operands and L2 is the result(where to jump)

Practice Questions-Quadruples

```
1. do
     {
        x=y+z-a;
     }while(a<b);

2.while(p<q and r>s)
   {
             if(p==1)
                        q=q+1;
             else

             while(p<=s)

             p=a+3;
}
```

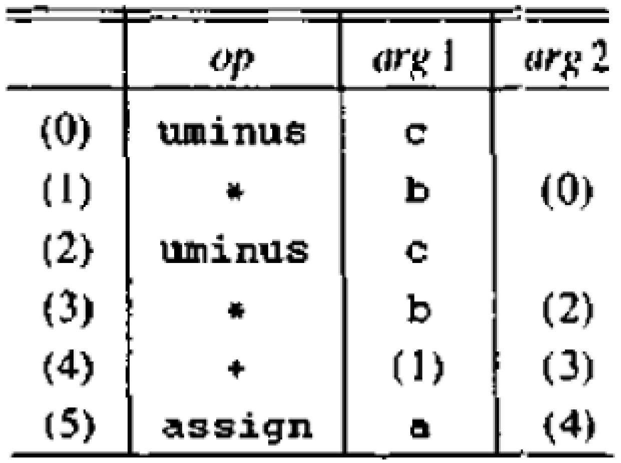
```
3. sum=0;
    for(i=1;i<=10;i++)
              sum=sum+a[j]+b[j];

4. switch(i+j)
   {
     case 1: x=y+z;
     case 2: p=q+r;
     default: u=u+v;
   }

5. switch(i+j)
   {
     case 1: x=y+z;
             break;
     case 2: p=q+r;
             break;
     default: u=u+v;
   }
```

Triples

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.
- The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure.
- Since three fields are used, this intermediate code format is as triples.



	op	*arg 1*	*arg 2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

Code for syntax tree

EXAMPLE 2

A ternary operation like x[i] : = y requires two entries in the triple structure while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

x[i] := y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

x := y[i]

Practice Questions

1. Construct the syntax tree and then draw the DAG for the statement

   e:= (a*b) + (c-d) * (a*b)

2. Write quadruples for the expression

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

3. Construct the Directed Acyclic Graph for the basic block given below and represent it in quadruple and triple forms.