Unit - III

SEMANTIC ANALYSIS, INTERMEDIATE CODE GENERATOR & SYMBOL TABLE

SEMANTIC ANALYSIS

- Attributed Grammars
- Syntax Directed Translation

INTERMEDIATE CODE GENERATOR

- Intermediate Forms of Source Programs - Abstract Syntax Tree, Polish Notation and Three Address Codes
- Intermediate Code Forms
- Type Checker

SYMBOL TABLE

- Symbol Table Format
- Organization for Block Structures Languages
- Hashing

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Three-Address Code

Outline:

- Addresses and Instructions
- Quadruples
- Triples

Three-Address Code

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like **x + y * z** might be translated into the sequence of three-address instructions:

$$t_1 = y * z$$

$$t_2 = x + t_1$$

where t_1 and t_2 are compiler-generated temporary names.

This multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization.

The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

Three-Address Code

Example:

Three-address code is a linearized representation of a **syntax tree** or a **DAG** in which explicit names correspond to the interior nodes of the graph. A **DAG** and its corresponding three-address code is shown below:

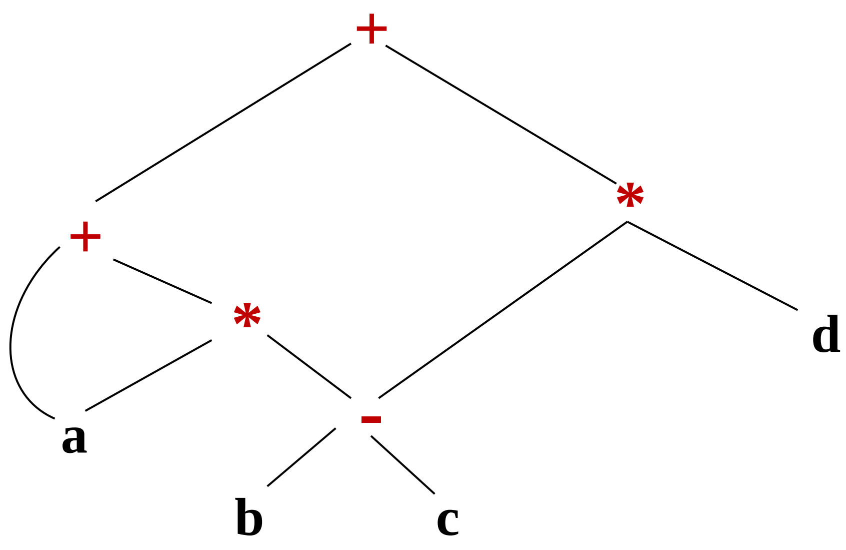

Fig: **DAG** for the expression `a + a *(b-c)+(b-c)*d`

Three-Address Code

Example:

Given expression: `a + a *(b-c)+(b-c)*d`

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

Fig: Three-address code

Addresses and Instructions

- Three-address code is built from two concepts: *addresses* and *instructions*.
- In object-oriented terms, these concepts correspond to *classes*, and the various kinds of addresses and instructions correspond to appropriate subclasses.
- *Alternatively*, three-address code can be implemented using records with fields for the addresses; records called **`quadruples`** and **`triples`**.

Addresses and Instructions

An ***address*** can be one of the following:

- A ***name***. For convenience, we allow source-program names to appear as addresses in **three-address code**. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- A ***constant***. In practice, a compiler must deal with many different types of constants and variables. Type conversions within expressions are considered.
- A ***compiler-generated temporary***. It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

Addresses and Instructions

List of the common ***three-address instruction forms***:

1. Assignment instructions of the form **`x = y op z`**, where **`op`** is a binary arithmetic or logical operation, and **`x`**, **`y`**, and **`z`** are addresses.

2. Assignments of the form **`x = op y`**, where **`op`** is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.

3. ***Copy instructions*** of the form **`x = y`**, where **`x`** is assigned the value of **`y`**.

Addresses and Instructions

4. An unconditional jump **`goto L`**. The three-address instruction with label **`L`** is the next to be executed.

5. Conditional jumps of the form **`if x goto L`** and **`ifFalse x goto L`**. These instructions execute the instruction with label **`L`** next if **`x`** is true and false, respectively. Otherwise, the *following(6)* three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as ***`if x relop y goto L`***, which apply a relational operator (**<**, **==**, **>=**, etc.) to **`x`** and ***`y`***, and execute the instruction with label **`L`** next if **`x`** stands in relation ***`relop`*** to ***`y`***. If not, the three-address instruction following ***`if x relop y goto L`*** is executed next, in sequence.

Addresses and Instructions

7. Procedure calls and returns are implemented using the following instructions: `param x` for parameters; `call p, n` and `y = call p, n` for procedure and function calls, respectively; and `return y`, where `y`, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

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

generated as part of a call of the procedure $p(x_1, x_2, ..x_n)$. The integer n, indicating the number of actual parameters in "`call p, n`," is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after `p` returns its value; that value becomes another parameter of the later call.

Addresses and Instructions

8. Indexed copy instructions of the form `x = y[i]` and `x[i]= y`. The instruction `x = y[i]` sets `x` to the value in the location `i` memory units beyond location `y`. The instruction `x[i]= y` sets the contents of the location `i` units beyond `x` to the value of `y`.
9. Address and pointer assignments of the form `x = &y`, `x = *y`, and `*x = y`. The instruction `x = &y` sets the `r-value` of `x` to be the location (`l-value`) of `y`. `l-value` and `r-value` are appropriate on the left and right sides of assignments, respectively. In the instruction `x = *y`, `y` is a pointer or a temporary whose `r-value` is a location. The `r-value` of `x` is made equal to the contents of that location. Finally, `*x = y` sets the `r-value` of the object pointed to by `x` to the `r-value` of `y`.

Addresses and Instructions

Example:

Consider the statement

```
do i = i+1;
while (a[i] < v);
```

Two possible translations of this statement are shown below:

Fig: Two ways of assigning labels to three-address statements

(a) Symbolic labels

L: t_1 = i + 1
i = t_1
t_2 = i * 8
t_3 = a [t_2]
if t_3 < v goto L

Addresses and Instructions

Example:

Fig: Two ways of assigning labels to three-address statements

(b) Position numbers

```
100:  t1 =  i + 1
101:  i =  t1
102:  t2 =  i * 8
103:  t3 = a [t2]
104:  if t3 < v goto L
```

Quadruples

- The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure.
- In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands.
- *Three* such representations are called

1. **Quadruples**
2. **Triples** and
3. **Indirect Triples**

Quadruples

- A ***quadruple*** (or just "***quad***") has four fields, which we call ***op***, arg_1, arg_2, and ***result***. The ***op*** field contains an internal code for the operator.
- For instance, the three-address instruction `x = y + z` is represented by placing `+` in ***op***, `y` in arg_1, `z` in arg_2, and `x` in ***result***.

The following are some exceptions to this rule:

1. Instructions with unary operators like `x = minus y` or `x = y` do not use arg_2. Note that for a copy statement like `x = y`, ***op*** is `=`, while for most other operations, the assignment operator is implied.
2. Operators like `param` use neither arg_2 nor ***result***.
3. Conditional and unconditional jumps put the target label in result.

Quadruples

Example: Three-address code and its quadruple representation

Three-address code for the assignment `a = b *- c + b *- c`

(a) Three-address code

t_1 = minus c or - c

t_2 = b * t_1

t_3 = minus c or - c

t_4 = b * t_3

t_5 = t_2 + t_4

a = t_5

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

(b) Quadruples

Triples

- A ***triple*** has only <u>three fields</u>, which we call ***op***, arg_1, and arg_2.
- <u>Note</u> that the ***result*** field in ***Quadruples*** is used primarily for temporary names.
- Using ***triples***, we refer to the result of an operation ***x op y*** by its position, rather than by an explicit temporary name.
- Thus, instead of the temporary t_1 in ***Quadruples***, a ***triple*** representation would refer to position **(0)**.
- Parenthesized numbers represent pointers into the ***triple*** structure itself.
- ***Triples*** are equivalent to ***signatures*** in "Algorithm-The value-number method for constructing the nodes of a **DAG**". Hence, the **DAG** and ***triple*** <u>representations of expressions</u> are equivalent.
- The equivalence ends with expressions, since syntax-tree variants and three-address code represent control flow quite differently.

Triples

Example:

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

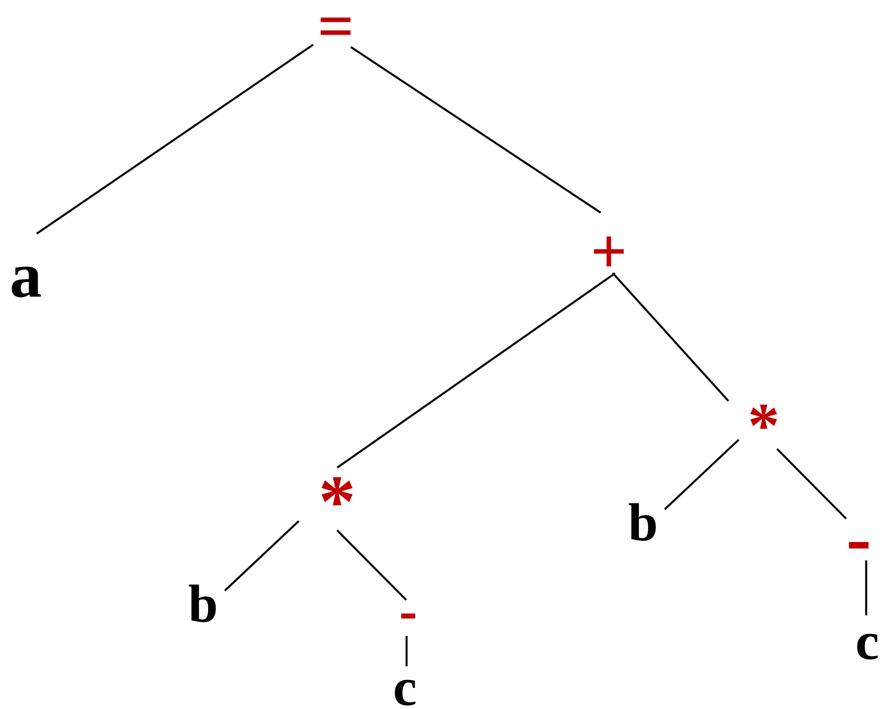

Fig: Syntax tree for the assignment `a = b *- c + b *- c`

Triples

Example: Three-address code and its triple representation for the assignment `a = b *- c + b *- c`

(a) Three-address code

t_1 = minus c or - c

t_2 = b * t_1

t_3 = minus c or - c

t_4 = b * t_3

t_5 = t_2 + t_4

a = t_5

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

(b) Triples

The copy statement a = t_5 is encoded in the triple representation by placing a in the arg_1 field and (4) in the arg_2 field.

t_1 (0); t_2 (1); t_3 (2); t_4 (3); t_5 (4)

Triples

- A benefit of ***quadruples*** over ***triples*** can be seen in an optimizing compiler, where instructions are often moved around.
- With ***quadruples***, if we move an instruction that computes a temporary ***t***, then the instructions that use ***t*** require no change.
- With ***triples***, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.

Indirect Triples

- ***Indirect triples*** consist of a listing of pointers to ***triples***, rather than a listing of triples themselves.
- For example, let us use an array instruction to list pointer to triples in the desired order. Then, the above ***triples*** might be represented as shown below:

	instruction
30	(0)
31	(1)
32	(2)
33	(3)
34	(4)
35	(5)

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

Fig. **Indirect Triples representation of three-address code**

Indirect Triples

- With ***indirect triples***, an optimizing compiler can move an instruction by reordering the ***instruction list***, without affecting the ***triples*** themselves.
- When implemented in **Java**, an array of instruction objects is analogous to an ***indirect triple*** representation, since **Java** treats the array elements as references to objects.

Practice Problems

EX-1. Translate the arithmetic expression **`a + -(b + c)`** into:

a) A syntax tree

b) Quadruples

c) Triples

d) Indirect triples

EX-2. Translate the following assignment statements into:

a) A syntax tree b) Quadruples c) Triples d) Indirect triples

i. **`a = b[i] + c[j]`**

ii. **`a[i] = b*c - b*d`**

iii. **`x = f(y+1) + 2`**

iv. **`x = *p + &y`**

Summary

Three-Address Code

- Addresses and Instructions
- Quadruples
- Triples

Reading: Aho2, Section 6.2.1 to 6.2.4

Next Lecture: Type Checker