Intermediate Code Generation

Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

Benefits

- Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, 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. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- Retargeting is facilitated
- It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.

Benefits (Cont.)

If we generate machine code directly from source code then for n target machine we will have n optimizers and n code generators but if we will have a machine independent intermediate code, we will have only one optimizer. Intermediate code can be either language specific (e.g., Bytecode for Java) or language independent (three-address code).

Representation

- Postfix Notation
- Three Address Code
- Syntax Tree
- Directed Acyclic Graph (DAG)

Postfix Notation

- The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b
- The postfix notation for the same expression places the operator at the right end as ab +.
- In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +.
- No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression.
- In postfix notation the operator follows the operand.

Example

- The postfix representation of the expression (a – b) * (c + d) + (a – b) :

ab – cd + *ab -+

Three Address Code

- They are generated by the compiler for implementing Code Optimization.
- They use maximum three addresses to represent any statement.
- They are implemented as a record with the address fields.

Common Three Address Instruction Forms

- Assignment Statement
- Copy Statement
- Conditional Jump
- Unconditional Jump
- Procedural Call

Assignment Statement

$$x = y \text{ op } z$$

$$x = \text{op } y$$

Here,

- x, y and z are the operands.
- op represents the operator.

It assigns the result obtained after solving the right side expression of the assignment operator to the left side operand.

Copy Statement

$$x = y$$

Here,

- x and y are the operands.
- = is an assignment operator.

It copies and assigns the value of operand y to operand x.

Conditional Jump

If x relop y goto X

Here,

- x & y are the operands, X is the tag or label of the target statement, relop is a relational operator.

If the condition "x relop y" gets satisfied, then-

- The control is sent directly to the location specified by label and all the statements in between are skipped.

If the condition "x relop y" fails, then-

- The control is not sent to the location specified by label X, the next statement appearing in the usual sequence is executed.

Unconditional Jump

goto X

Here, X is the tag or label of the target statement.

On executing the statement,

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

Procedural Call

param x call p return y

Here, p is a function which takes x as a parameter and returns y.

Three Adress Code (Problem 1)

Write Three Address Code for the expression given below

$$a = b + c + d$$

Solution:

$$T1 = b + c$$
$$T2 = T1 + d$$
$$a = T2$$

Three Adress Code (Problem 2)

Write Three Address Code for the expression given below-

$$-(a \times b) + (c + d) - (a + b + c + d)$$

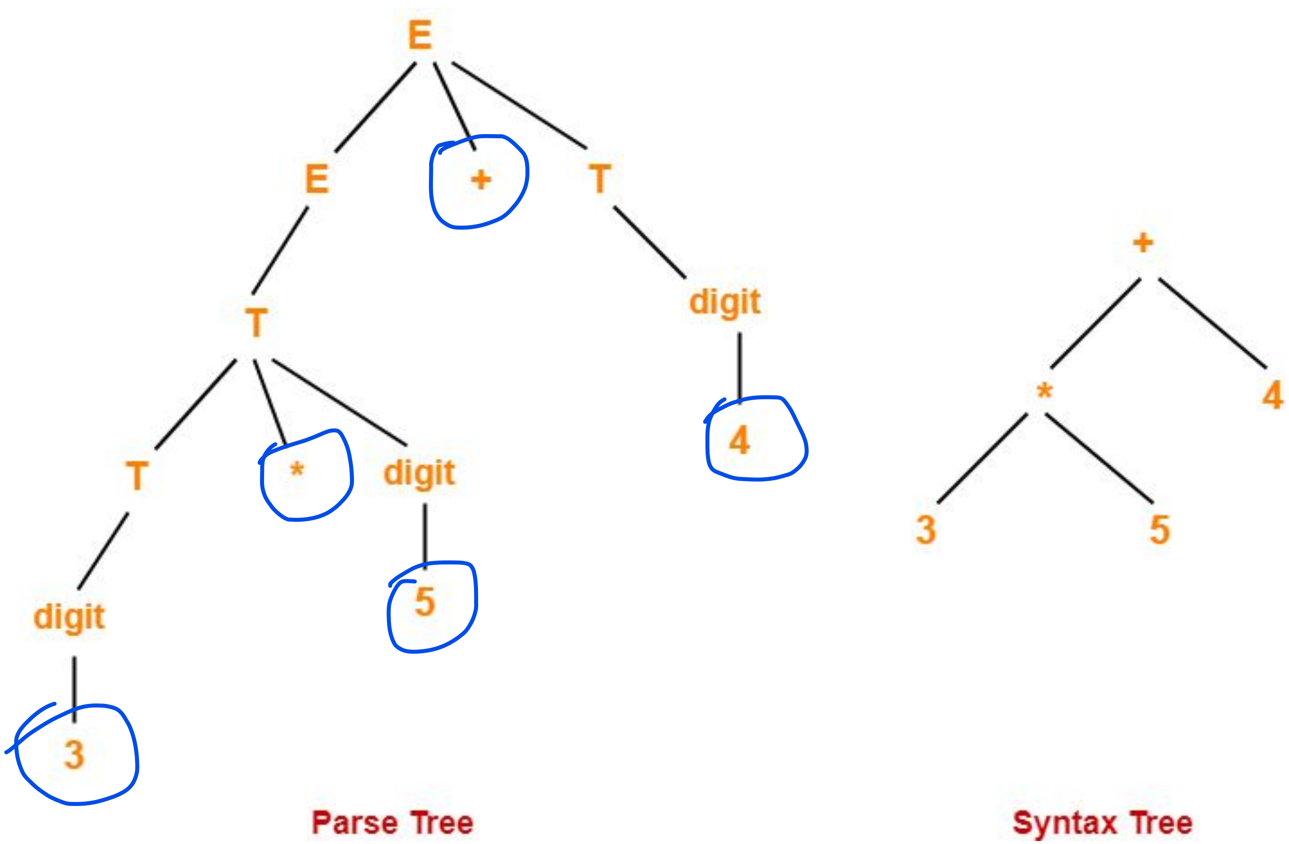
Solution:

T1 = a x b
T2 = uminus T1
T3 = c + d
T4 = T2 + T3
T5 = a + b
T6 = T3 + T5
T7 = T4 – T6

Syntax Tree

- Syntax tree is nothing more than condensed form of a parse tree.
- The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link.
- In syntax tree the internal nodes are operators and child nodes are operands.
- To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Syntax Tree (Example)



Directed Acyclic Graph

- In compiler design, a Directed Acyclic Graph (DAG) is a special kind of abstract syntax tree (AST) where a unique node is present for every unique value.

- In other words, there are no two nodes which have the same value.

Rules

- Interior nodes always represent the operators.
- Exterior nodes (leaf nodes) always represent the names, identifiers or constants.
- A check is made to find if there exists any node with the same value.
- A new node is created only when there does not exist any node with the same value.
- This action helps in detecting the common sub-expressions and avoiding the re-computation of the same.
- The assignment instructions of the form x:=y are not performed unless they are necessary.

DAG (Example)

1

- Consider the given expression and construct a DAG for it-

(a + b) x (a + b + c)

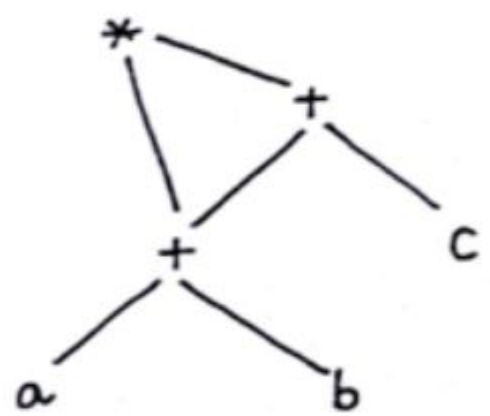
Three Address Code

T1 = a + b

T2 = T1 + c

T3 = T1 x T2

DAG



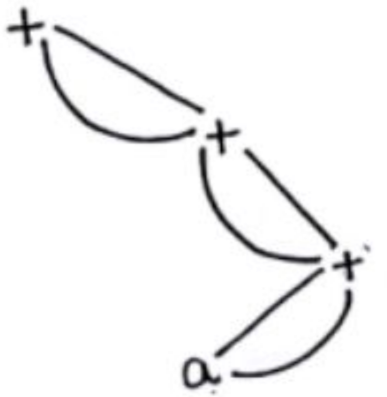
DAG (Example)

2

- Consider the given expression and construct a DAG for it-

$$(((a + a) + (a + a)) + ((a + a) + (a + a)))$$

Solution:



DAG (Example)

3

$T_1 = a + b$
$T_2 = a - b$
$T_3 = T_1 * T_2$
$T_4 = T_1 - T_3$
$T_5 = T_4 + T_3$