gradeup

Code Generation and Code Optimization

Code Generation and Code optimization:

Content:

1. **Types Expression**
2. **Writing simple type checker**
3. **Intermediate code generation**
4. **Intermediate form**
5. **Syntax tree**
6. **Three address code**
7. **Code optimization**
8. **Compile time evaluation**
9. **Dead code evaluation**
10. **Code movement**
11. **Code generation**

Types Expressions: It is a convenient way to express the type of a language construct. Ait may be defined as follows:

1. It may be a basic type. Thus int, float, char, double are type expressions.
2. T-error is a special basic type which indicates an error during type checking
3. Null is another basic type which indicates that there is no type attached to a statement.
4. A name of a type is a type expression.
5. A type expression may be formed by applying an operator called a constructor to other type expression. These constructors include the array, product, struct, pointer and fuction.

Writing a simple type checker: It checks for type compatibility which essentially finds out whether two type expressions are equivalent. This type equivalence may be name equivalence or structural equivalence.

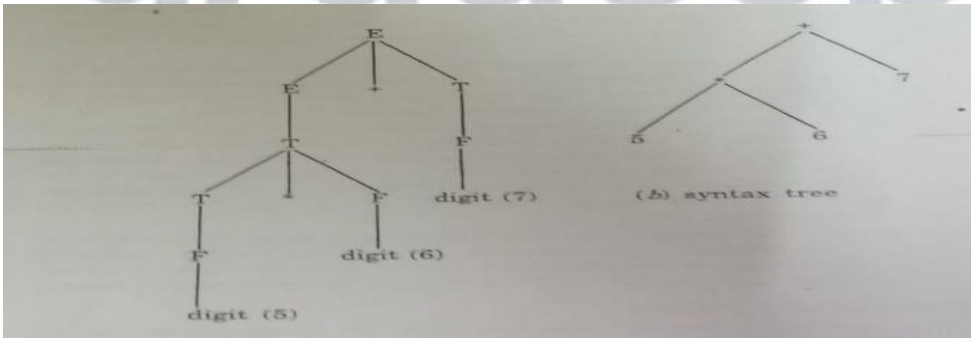
Intermediate code generation: compiler often generate intermediate code instead of translating an input file directly to binary. We have seen that it is not always possible to generate target code for a program in one pass of the compiler due to various possible reasons which are as follows

1. Sufficient core memory may not be available to accommodate a single pass compiler.
2. A multi pass structure may be required to satisfy the primary aims of the compiler to generate a highly efficient target code or to occupy minimum possible storage space.

Intermediate Forms: There are several intermediate code forms viz. postfix notation, syntax trees and three address code. The choice of intermediate code form to be used in a compiler depends on two important strategy. These are:

1. Ease of conversion from the source program to the intermediate code and.
2. Ease of subsequent processing which is to be done on intermediate code viz. generation of target code in the simplest manner or program optimization.

Syntax Trees: It is a condensed form of parse tree which is used to represent the syntactic structure of the language constructs. Operators and keywords are associated with the interior nodes in a syntax tree and do not appear as leaves.



Construction of syntax trees: A syntax tree an expression may be constructed in a bottom-up fashion by first creating the leaf nodes for the identifiers and constants appearing in the expression and then constructing the sub trees for the sub expressions. An operator node may be created as the root with the operand nodes as

its children. These child nodes nay represent the sub expression which is constituting the operands of that operator.

Three Address Code: In three address code, each statement generally contains 3 addresses, two for the operands and one for the result. This is why it is termed as three address code. It is a sequence having a general form

a: b op c

where op is any operator and a, b and c can be variables, constants or the temporary variables generated by the compiler. We can say that three address code is a linearized representation by the following three address code.

t_1 : = b * 9

t_2 : = a + t_1

The commonly used three-address statements are as follows

1. a := b op c, an assignment statement containing a binary operator op which can be arithmetic or logical operator.
2. a := op b, an assignment statement containing a unary operator op which can be unary minus, logical negation, conversion operator or shift operator.
3. a := op b, a copy statement which copies the contents of b to a.
4. goto X, an unconditional jump which causes the control flow to execute the three address instruction with label X as the next instruction.
5. If a relop b goto X, a conditional jump which applies a relational operator, relop, to a and b, and executes the statement with label X, if the relation is true. If not, the next statement in the normal flow sequence is executed.
6. If a goto X, a conditional jump which executes the statement with label X, if a is zero. If not, the next statement in the normal flow sequence is executed.
7. Param a, call p,n and return x, for procedure calls. n indicates the number of parameters in the call to p.x indicates a returned value and is optional.

Implementation of the three address code: It can be implemented as records having fields for the operators and operands. It can be realized in several ways viz. quadruples and triples.

Triples: It is a record structure having three fields as follows

Operator	Operand1	Operand2

Quadruples: It is a record structure having four fields as follows

Operator	Operand1	Operand2	result

Where operand1 and operand2 are two operands for the operator and result field contains the result of the operation on operand1 and operand2.

For example a : =b + c will be represented as

+b c a

Type conversion: In the above generation of three address code for assignment statements, we assumed that all the operands are of the same type which is not always possible. In the practice, programming languages allows certain operations on mixed types. For example C allows a∗b, where a is of type real and b is of type integer. In such cases, the compiler may have to first convert one of the operands to ensure that both operands are of the same type before generating appropriate code.

It is called coercion, if it is implicit context dependent type conversion done automatically by the compiler. A type conversion is explicit if it is done by writing a function in the program.

Code Optimization: the term optimization may be applied to a technique designed to obtain more efficient object code than would be obtained by simple, straight forward code generators. Two constraints on the techniques used to perform code optimizations are

- They must ensure that the transformed program is semantically equivalent to the original program.
- The improvement of the program efficiency must be achieved without changing the algorithms which are used in the programs.

The input program is generally written in a high level language and the output program can be in a high level language or in a low level language. If both input and output programs are in high level language, the optimizer is known as source to source optimizer. If the output is in low level language, the optimizer is known as an optimizing compiler.

The main goal of optimization is to produce target program with high execution efficiency. To obtain the best results in a minimum effort we can identify the frequently executed parts of a program and make them as efficient as possible. Most programs spend maximum percentage of their execution time in a small fraction of a program. If we can optimize this small fraction of a program, efficiency may be improved drastically. It may be obtained by rearranging the computations in a program without changing the meaning of the program.

Optimization can be done at each level as follows:

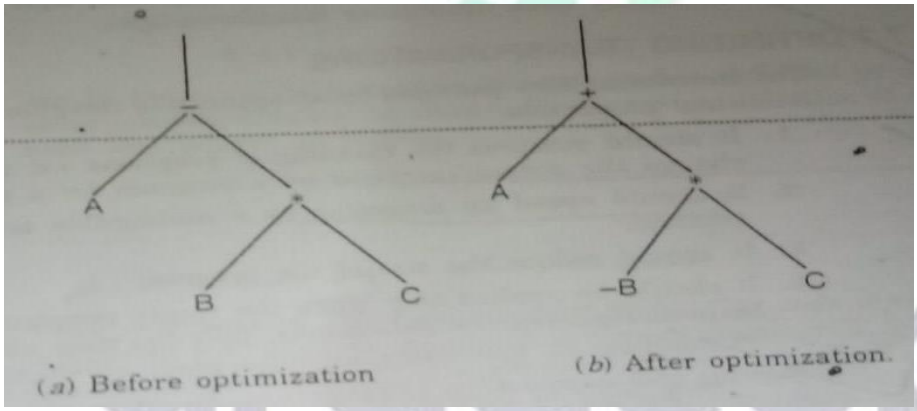
- At the source level by the user who can change algorithm or transform loops.
- At the intermediate code level by the compiler which can improve loops, improve address calculations.
- At the target code level by the compiler which can keep the most heavily used variables in registers.

Machine Dependent/ Independent Optimization: It can be classified as machine dependent or machine independent. Machine dependent optimizations exploit characteristics of the target machine in terms of addressing structures and instructions set to reduce the amount of code of execution time.

Properties of the operations involved

- Allocation of scare resources to achieve high efficiency in program execution. Example the top of stack registers, few arithmetic registers etc.
- Use of immediate instructions where a value is a part of the instruction, incrimination where a memory location can be incremented by some constant, indexing or indirection and vector operations are some special machine features that can be gainfully exploited.
- If data is intermixed with the instruction sequence, it can be accessed more efficiently on some machines.

Let us consider which illustrates a situation in which we can optimize the code by using a LOAD COMPLEMENT instruction.



Machine Independent optimization: they are based on the mathematical properties of a sequence of source statements. An optimization essentially means analyzing the overall purpose of the statement sequence and finding an equivalent sequence which can be translated to the minimum code.

Optimizing Transformations: They are provided by an optimizing compiler should have following properties.

- It should preserve the meaning of programs i.e. it should not change the output produced by a program for a given input.
- It should speed up programs by a measurable amount on an average.
- It should reduce the size of the program

Common sub expression elimination: An expression need to be evaluated if it was previously computed and the values of variables in this expression have not changed since the earlier computation.

Example , consider the following code

a := b ** c;

d := b** c+ x-y

we can eliminate the second evaluation of b**c from this code if none of the intervening statements has changed its value. We can thus, rewrite the code as

t_1 :=b** c

a=t_1

d := t_1 + x-y

Let us consider the following code

a := b ** c

b := x;

d := b** c +x - y

we can say that the two expressions are common if

1. They are lexically equivalent i.e. they consist of identical operands connected to each other by identical operators.
2. They evaluate to identical values i.e. no assignment statements for any of their operands exist between the evaluations of these expressions.
3. The value of any of the operands used in the expression should not be changed even due to a procedural call, which might change it as a side effect.

Compile-time evaluation: we can improve the execution efficiency of a program by shifting execution time actions to compile time so that they are not performed repeatedly during program execution. We can evaluate an expression with constant operands at compilation time and replace that expression by a single value. This is called folding.

Dead code elimination: If the value contained in the variable at a point is not used anywhere in the program subsequently, the variable is said to be dead at that place.

Code Movement: The motivation for performing code movement in a program is to improve the execution time of the program by reducing the evaluation frequently of expressions. This can be done by moving the evaluation of an expression to other parts of the program. Let us consider an example

If a<10

b= x 2 - y 2;

Else

b=5;

a=(x 2 - y 2) * 10;

Strength reduction: In frequency reduction transformations we tried to reduce the execution frequency of expressions by moving the code. There is other class of transformations which perform equivalent actions indicated in a source program by reducing the strength of the operators. By strength reduction, we mean replacing the high strength operator by a low strength operator.

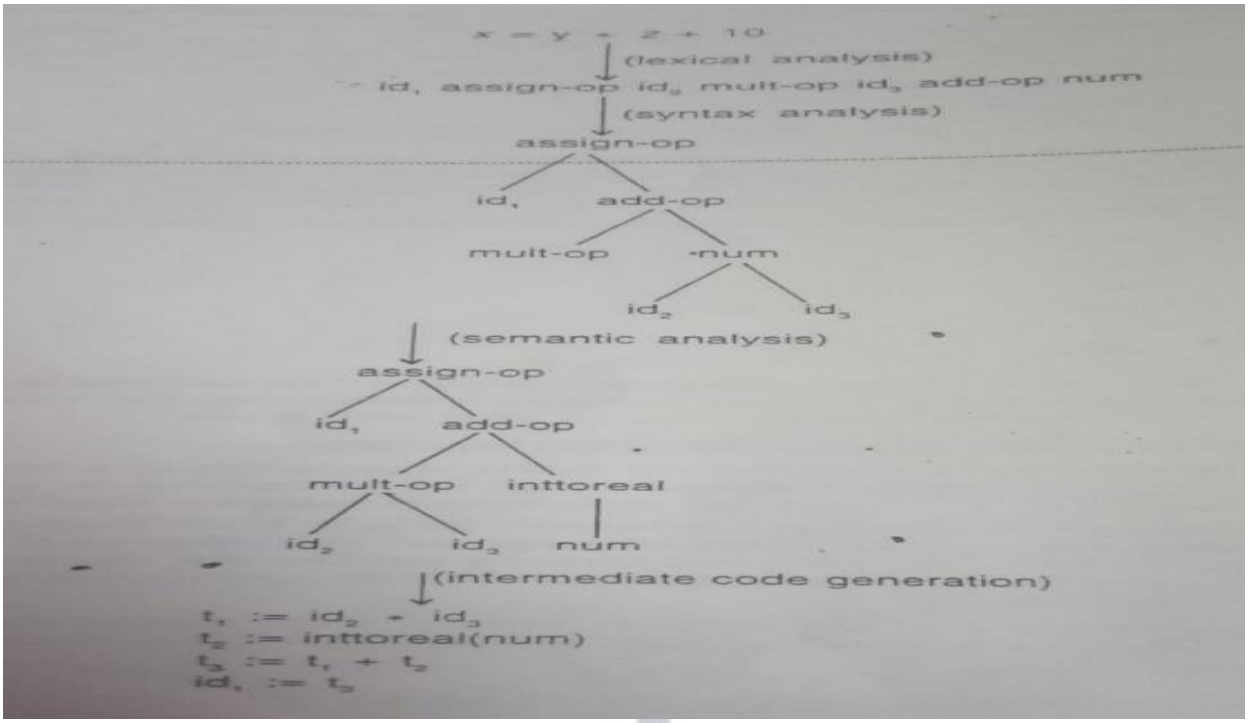
Loop test replacement: We can replace a loop termination test phrased in terms of one variable by an equivalent loop test phrased in terms of another variable.

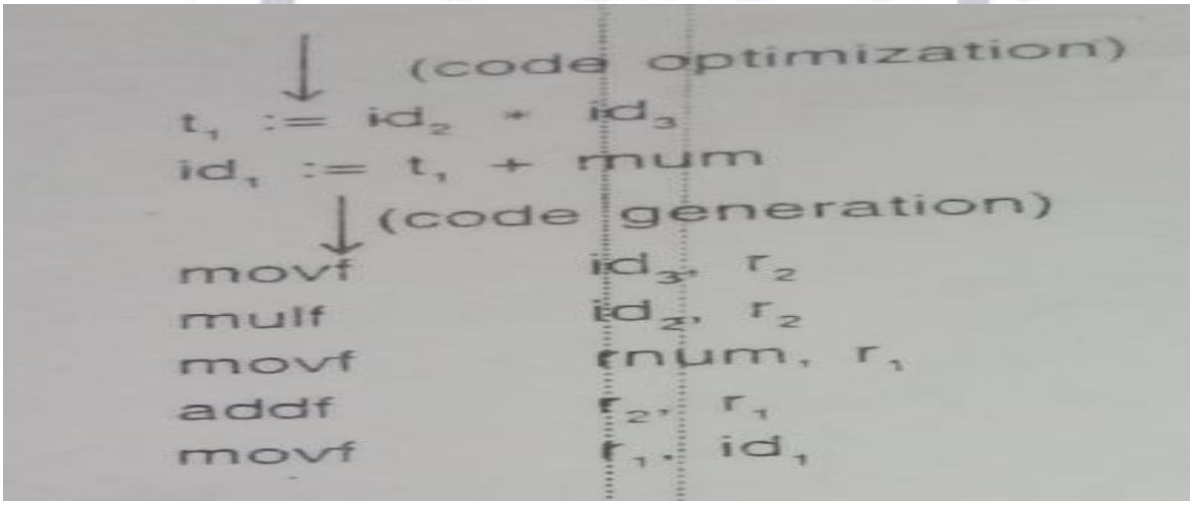
Code Generation:

It can be considered as the final phase of compilation. Through post code generation, the optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. Code generated by the compiler is an object code of some lower-level programming language, i.e. assembly language. We have seen that source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

a. It should carry the exact meaning of the source code.

b. It should be efficient in terms of CPU usage and memory management.





gradeup SUPER

Gradeup UGC NET
Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now

gradeup.co