# 浙江大学


课程考查报告

**题　　目**：关于编译器及编译器优化的一些思考与探究.

**课程名称**：《编译原理》.

**指导老师**：王强.

**姓　　名**：.

**学　　号**：.

2020-06-26

# Abstract

After a semester of learning about the *Compile Principle* course, I have learned a lot about the compiler, including how it works and how it is implemented. The knowledge about the compile is not limited to those in class and there are still a lot of problems in relational fields.

In this report, I want to display something about the compiler, including what a compiler is, what I learned in the course, and something else, mainly about the optimization techniques. In the end, it's some of my thoughts and feelings about this course.

The review part of this course is written with my understanding and experience on this course. It is also a summary of mine about the textbook. The additional optimization part is written by reading *Compilers Principles, Techniques, & Tools* (**the dragon book**). I find this bool introduce the compiler more detailly in the code optimization part, which take up not a very large part of the course.

Limited to the length of the article (want to make this report not too long to read) and time, some details are omitted and the main purpose of this report is introducing, but not explaining or arguing.

**Keys**: Programming languages, Compiler, Compiler optimization

# Ⅰ. About Programming Languages

Programming languages supply much convivence for humans to use computers. At the beginning, people can only write machine codes for computers to read. The assembly language makes codes more understandable for people to read, but it is still just another form of machine codes and satisfies only specific machines.

Machine languages and assembly languages is designed for underlying hardware and not so friendly to humans' thinking methods. Besides, since there are many kinds of computer hardware, if some program is required to satisfy most machine environment, its source code may need to be written many times with different languages. This causes a lot of troubles and work for programmers.

Then some high-level languages are created to improve the developing environment and make it easier to program. The high-level languages are designed based on humans' way to thinking. Removing some mechanisms that satisfy the hardware environment, like finite registers and single-type variables, makes programing more free and closer to natural languages. An outstanding representative is the C language.

Nowadays, there are more than hundreds of programming languages in the world, but the C language is still the most popular one. C language allows programmers to write codes in understandable grammar with respect to humans' habits, and some underlying mechanisms are remained, like pointer, which can supply free coding experience with ensuring the efficiency of

the programs.

There are two main types of high-level languages, interpreted languages and compiled languages. An interpreter can interpret an interpreted language and run it line by line, but the program always runs slowly because the interpreter won't interpret the whole program once. A compiler can translate a compiled language into some low-level language that can be read and run by machines, so programs produced by compiling always run faster than those produced by interpreting. Of course, interpreters perform better than compilers on the checking and reporting of errors.

A compiler is a computer program with a source code as its input and outputs a new code with another language (target code). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

The compile procedure can be complex and divided into several steps. How to build a compiler and optimize it is an important problem. Many masters or professors work on this and have make huge contributions.

## II . The Review of the Course

This part is mainly about the textbook and what mentioned in the classes.

Before the compiler processes the source codes, there is a program called preprocessor to combine the dispersed source codes into a complete one. The preprocessor also replaces and unfolds all macros.

We can recognize a compiler as a black box. It maps a source code into an equivalent object program. In fact, it has a lot of detailed structures. We can open the black box can divide the structures into two main parts, a part to analyze and a part to synthesize.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

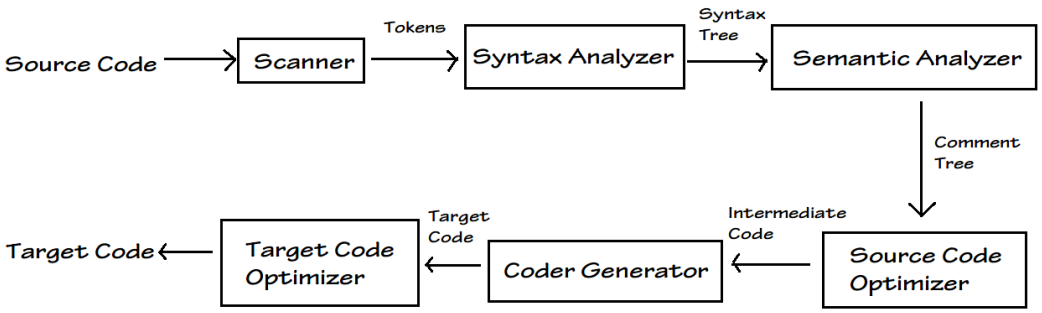
The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

The analysis part is often called the front end of the compiler; the synthesis part is the back end.

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. In practice, several phases may be grouped together, and the intermediate representations between

the grouped phases need not be constructed explicitly.

Usually, the phases can be divided as following: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and target code generation. Many compilers do some machine-independent optimization in code generation phase and it's an optional work.




### 1. Lexical Analysis

This is the first phase of compiling, and it is also called scanning. Just like its name, in this phase, the lexical analyzer, or the scanner, reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. The scanner identifies all tokens with some specific rules. Tokens can be used in syntax analysis as basic elements and construct syntax factors.

A token can have its attributes, like arithmetic value or variable type. The specific attribute contents of tokens can be used to calculate the attributes of higher-level syntax elements.

### 2. Syntax Analysis

The second phase of a compiler is called syntax analysis, or parsing. The syntax analyzer parses the relations between tokens and produce a syntax structure representative. A common method to represent the syntax structure is to use the syntax tree. Every internal node of the syntax tree represents some operation, and its child nodes represent the corresponding operators. In this phase, all syntax information will be written into a syntax tree and we can make further analysis based on that.

The grammar rules of programming languages are usually context-free. There are two main methods to parse such grammar, top-down parsing and bottom-up parsing, which are both recursive. The detailed parsing method can be LL(1) if top-down parsing is used, or LR(0), LR(1), LR(1), LALR(1) if bottom-up parsing is used.

Similar to the lexical analysis, the theory about the finite automation is applied in analysis but there are some differences. In lexical analysis, the finite automation is used to represent the states when analyzing the regular formulas and matching them with character streams, while in syntax analysis, it is used to represent the states of derivations comparator and match the token combinations with derivations or other syntax structures.

A point in syntax analysis is how to eliminate the ambiguities in the text, such as the hanging else problem. We can define some rules when conflicts occur in matching, or modify the

derivations into other forms to avoid such phenomenon.

## 3. Semantic Analysis

When we obtain a syntax tree, or an abstract syntax tree from the syntax analyzer, we have already known the syntax structure of the program. In this phase, the semantic analyzer will calculate the additional information required in the compile procedure, that is, figuring out what the program really want to do, and what steps would the program execute in order if it receive some specific inputs.

One important thing to do is to compute the attributes of each node in the abstract syntax tree. For example, expressions constitute a large part of the program, and computing the value of expressions is an important thing since an expression without value almost loses its meaning of existence. Different nodes may have different attributes to compute and check the type of the node is necessary before computing. In fact, the type of nodes can be denoted in the syntax analysis phase as long as give a node its name when it is matched in some syntax structure.

The attributes can be divided into two types, synthesized attributes and inherited attributes. We begin the semantic analysis with the recursive traversal of the syntax tree and computing attributes of nodes when visited them. The synthesized attributes of a node can be computed out from its child nodes while the inherited attributes can be computed out from its ancient nodes and sibling nodes.

The symbol table is an important data structure in the semantic analysis phase. It has associations with the syntax analyzer, even the scanner. The symbol table stores the information of variables and functions. Usually the symbol tables can be designed like a linked list or a stack so that global variables and local variables can be distinguished, and the variable scope can have more layers. If two variables with the same name appear in one program, the one in the innermost layer will shield the other. This is also a rule to handle the ambiguity in variable using.

One of the main tasks of a compiler is the calculation and the maintenance of data type information and the use of this information to ensure that each part of the program is meaningful under the effect of the language's type rules. Usually, these two tasks are closely related and executed together, that is, when the compiler does type checking.

Data type information can be static or dynamic or a mixture of both. In dynamic languages, the compiler must generate code at execution time to complete type inference and type checking. However, in static languages, the type information is mainly static, and the correctness check is mainly performed before the program is executed. Static type information is also used to determine the memory size and memory access method required for each variable allocation, which can be used to simplify the operating environment.

Usually, the compiler checks types when using an array, including the element type of the array and the index type, when processing a variable of user-defined type like the struct variable in C language, and when a function is declared of called, including the return type and the types of parameters. In C language, type check should also be executed when using a pointer.

In most object-oriented languages, there are class descriptions, except for the operation

descriptions it contains, which are called methods or member functions. Class descriptions can create or not create new types (created in C++) in an object-oriented language. Even so, class descriptions are not just types, because they allow the use of features belonging to the type system, such as inheritance and dynamic binding. These later features must be maintained through independent data structures, such as class hierarchy (direct acyclic graph), to achieve inheritance, and virtual method table (virtual method table), to achieve dynamic binding.

Besides, if the source language has some advanced characteristics like overload, we also need to take it into the consideration scope of type checking.

**4. Code Generation**

The final task of translation is to generate the executable code that can be run on the target machine. This is the most complex phase in the compile procedure, since it relies not only the characteristics of the source language, but also the target machine and the running environment, even the details of the operation system of the target computer.

To simplify this complex procedure, we can divide the code generation phase into several steps, including the intermediate code generation.

The intermediate code is something closer to the target code, similar to the assembly language in form but not the same as. The purpose of generating the intermediate code is to give the syntax tree a representation that is closer to the target code. The compiler can generate the target code easily from the intermediate code. Since the intermediate code is machine-free, which means the compiler only needs to consider how to translate the intermediate code into different target codes, it saves a lot of times and make it more convenient for the compiler to adapt to different machine environments.

The intermediate code can have many forms, almost as the kinds of the compiler. But all intermediate codes represent some linearization of the syntax tree and because of this, the generation of the intermediate code usually be done with the semantic analysis. There are two general forms of the intermediate code: the three-address code and the P-code. The three-address code is designed as the form of calculating expressions while the P-code is designed like operating on a stack.

**5. Runtime Environment**

The runtime environment means the register and memory structure of the target computer, which is used to manage the memory and store the information required to guide the execution process. The runtime environment is decided by the source language, but not the target computer or the operation system.

In fact, almost all programming languages use one of the three types of runtime environment, and its main structure does not depend on the specific details of the target machine. The three types of environments are: fully static environment features of FORTRAN77, stack-based environments like C, C++, Pascal, and Ada, and the fully dynamic environments of function languages like LISP. The mixed form of these three types is also possible.

Another problem to consider is how the function caller passes the parameters to the callee.

There are two commonly used parameter passing mechanisms, pass by value and pass by reference.

In passing by value, arguments are expressions that are calculated at the time of the call, and when the process is executed, their values become the values of the parameters. This is the only parameter passing mechanism available in C language.

In reference passing, the argument must change with the assigned address (at least in principle). Instead of passing the value of the variable, the reference passes the address of the variable, so that the parameter becomes an alias for the independent variable, and any changes that occur on the parameter will appear on the independent variable. In FORTRAN77, reference passing is the only parameter passing mechanism. In Pascal, the reference is passed by using the "var" keyword, while in C++, the special character "&" is used in the parameter description to declare that the parameter is passed with reference.

Besides, there are another two parameter passing mechanisms that are important as well but not commonly used like the previous two, pass by value-result and pass by name.

Except not establishing a real alias, the result of the value-result transfer mechanism is similar to the reference transfer: copy and use the value of the argument in the process, and then when the process exits, copy the final value of the parameter back to the address of the argument. Therefore, this method is sometimes referred to as copy in, copy out, or copy storage. This is Ada's in out parameter mechanism.

Passing by name is the most complicated of the transfer mechanism. Since the idea of name transfer is that the argument is not assigned until the called program actually uses the argument (as a parameter), it is also called delayed evaluation.

The explanation of the passing by name is as follows: the text of the argument at the call point is regarded as the function on its right, and it is calculated whenever the corresponding parameter name is reached in the code of the called procedure. The program always calculates the arguments in the environment of the calling procedure, and always execute the procedure in the definition environment of the procedure.

# III. Local Machine-independent Optimizations

High-level language constructs can introduce substantial run-time overhead if we naively translate each construct independently into machine code. In the course of this semester, the phase of code optimization is not discussed in detail, but the optimization of code is an important part in compiler designing and we can do a lot of things to make our compiler more efficient. The optimization discussed should be independent with specific machine, that is, the optimization can work on any machine the compiler supports and the procedure of optimization involves no information about the target machine.

Of course, there exists optimization that aims to the target code, but it should be discussed with specific target code and has fewer universal meanings.

## 1. The Optimization of Basic Blocks

The concept of the basic block is defined as the maximal sequences of consecutive three-address instructions with the properties that

a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block $B$ to block $C$ if and only if it is possible for the first instruction in block $C$ to immediately follow the last instruction in block $B$. There are two ways that such an edge could be justified:

a) There is a conditional or unconditional jump from the end of $B$ to the beginning of $C$.

b) $C$ immediately follows $B$ in the original order of the three-address instructions, and $B$ does not end in an unconditional jump.

Then $B$ is a predecessor of $C$, and $C$ is a successor of $B$.

Usually we add two nodes, called the $entry$ and $exit$, that do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program. If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself.

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph) and the DAG representation of the basic block enable us to make some optimization on the codes in the basic block.

a) We can eliminate local common subexpressions, that is, instructions that compute a value that has already been computed.

b) We can eliminate dead code, that is, instructions that compute a value that is never used.

c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.

d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

Common subexpressions can be detected by noticing, as a new node $M$ is about to be added,

whether there is an existing node $N$ with the same children, in the same order, and with the same operator. If so, $N$ computes the same value as $M$ and may be used in its place. This technique was introduced as the "value-number" method of detecting common subexpressions.

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

There are several algebraic methods to optimize the basic block, like using algebraic identities, reduction in strength, and constant folding. The DAG-construction process can help us apply these and other more general algebraic transformations such as commutativity and associativity. For example, suppose the language reference manual specifies that $*$ is commutative, that is, $x * y = y * x$. Before we create a new node labeled $*$ with left child $M$ and right child $N$, we always check whether such a node already exists. However, because $*$ is commutative, we should then check for a node having operator $*$, left child $N$, and a right child $M$.

After we perform whatever optimizations are possible while constructing the DAG or by manipulating the DAG once constructed, we may reconstitute the three-address code for the basic block from which we built the DAG. For each node that has one or more attached variables, we construct a three-address statement that computes the value of one of those variables.

We prefer to compute the result into a variable that is live on exit from the block. However, if we do not have global live-variable information to work from, we need to assume that every variable of the program (but not temporaries that are generated by the compiler to process expressions) is live on exit from the block. If the node has more than one live variable attached, then we have to introduce copy statements to give the correct value to each of those variables. Sometimes, global optimization can eliminate those copies, if we can arrange to use one of two variables in place of the other.

**2. Peephole Optimization**

While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure. Nevertheless, many simple transformations can significantly improve the running time or space requirement of the target program.

A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole

optimization that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit.

There several applications of peephole optimization, like redundant instruction elimination, unreachable code elimination, flow-of-control optimization, algebraic simplification and strength reduction.

# IV. Global Machine-free Optimization

In global code optimization, code improvements take into account what happens across basic blocks. Most global optimizations are based on data-flow analyses, which are algorithms to gather information about a program. The results of data-flow analyses all have the same form: for each instruction in the program, they specify some property that must hold every time that instruction is executed. The analyses differ in the properties they compute. For example, a constant-propagation analysis computes, for each point in the program, and for each variable used by the program, whether that variable has a unique constant value at that point. This information may be used to replace variable references by constant values, for instance. As another example, a liveness analysis determines, for each point in the program, whether the value held by a particular variable at that point is sure to be overwritten before it is read. If so, we do not need to preserve that value, either in a register or in a memory location.

**1. The Principal Sources of Optimization**

There are many redundant operations in a typical program. Sometimes the redundancy is available at the source level. For instance, a programmer may find it more direct and convenient to recalculate some result, leaving it to the compiler to recognize that only one such calculation is necessary. But more often, the redundancy is a side effect of having written the program in a high-level language.

**2. Data-Flow Analysis**

"Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths. For example, one way to implement global common sub expression elimination requires us to determine whether two textually identical expressions evaluate to the same value along any possible execution path of the program. As another example, if the result of an assignment is not used along any subsequent execution path, then we can eliminate the assignment as dead code. These and many other important questions can be answered by data-flow analysis.

The execution of a program can be viewed as a series of transformations of the program state, which consists of the values of all the variables in the program, including those associated with stack frames below the top of the run-time stack. Each execution of an intermediate-code statement transforms an input state to a new output state. The input state is associated with the program point before the statement and the output state is associated with the program point after the statement. When we analyze the behavior of a program, we must consider all the

possible sequences of program points ("paths") through a flow graph that the program execution can take. We then extract, from the possible program states at each point, the information we need for the particular data-flow analysis problem we want to solve. In more complex analyses, we must consider paths that jump among the flow graphs for various procedures, as calls and returns are executed. However, to begin our study, we shall concentrate on the paths through a single flow graph for a single procedure.

There are three classic instances of data-flow problems: reaching definitions, live variables, and available expressions. The definition of each problem is given by the domain of the dataflow values, the direction of the data flow, the family of transfer functions, the boundary condition, and the meet operator.

| | Reaching Definitions | Live Variables | Available Expressions |
|---|---|---|---|
| Domain | Sets of definitions | Sets of variables | Sets of expressions |
| Direction | Forwards | Backwards | Forwards |
| Transfer function | $gen_B \cup (x - kill_B)$ | $use_B \cup (x - def_B)$ | $e\_gen_B \cup (x - e\_kill_B)$ |
| Boundary | $\text{OUT}[\text{ENTRY}] = \emptyset$ | $\text{IN}[\text{EXIT}] = \emptyset$ | $\text{OUT}[\text{ENTRY}] = \emptyset$ |
| Meet ($\wedge$) | $\cup$ | $\cup$ | $\cap$ |
| Equations | $\text{OUT}[B] = f_B(\text{IN}[B])$<br>$\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$ | $\text{IN}[B] = f_B(\text{OUT}[B])$<br>$\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$ | $\text{OUT}[B] = f_B(\text{IN}[B])$<br>$\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$ |
| Initialize | $\text{OUT}[B] = \emptyset$ | $\text{IN}[B] = \emptyset$ | $\text{OUT}[B] = U$ |

There several basic questions about data-flow algorithm formally:

1) Under what circumstances is the iterative algorithm used in data-flow analysis correct?

2) How precise is the solution obtained by the iterative algorithm?

3) Will the iterative algorithm converge?

4) What is the meaning of the solution to the equations

And there is a general approach that answers all these questions, once and for all, rigorously, and for a large family of data-flow problems. To make coding effort reduced and reduce programming errors with similar details, we can establish a common theoretical framework with the concept of the solution to normal data-flow problems. More details are omitted here, to make this report tidier and more readable.

We can also prove that there exist iterative algorithms for general frameworks that are convergent and work well.

### 3. Constant Propagation

Constant propagation is a forward data-flow problem. The set of data-flow values is a product lattice, with one component for each variable in a program. The lattice for a single variable consists of the following:

1) All constants appropriate for the type of the variable.

2) The value $NAC$, which stands for not-a-constant. A variable is mapped to this value if it is determined not to have a constant value. The variable may have been assigned an input value, or derived from a variable that is not a constant, or assigned different constants along different paths that lead to the same program point.

3) The value $UNDEF$, which stands for undefined. A variable is assigned this value if nothing may yet be asserted; presumably, no definition of the variable has been discovered to reach the point in question.

$NAC$ and $UNDEF$ are not the same and in fact they are essentially opposites. $NAC$ says we have seen so many ways a variable could be defined that we know it is not constant while $UNDEF$ says we have seen so little about the variable that we cannot say anything at all.

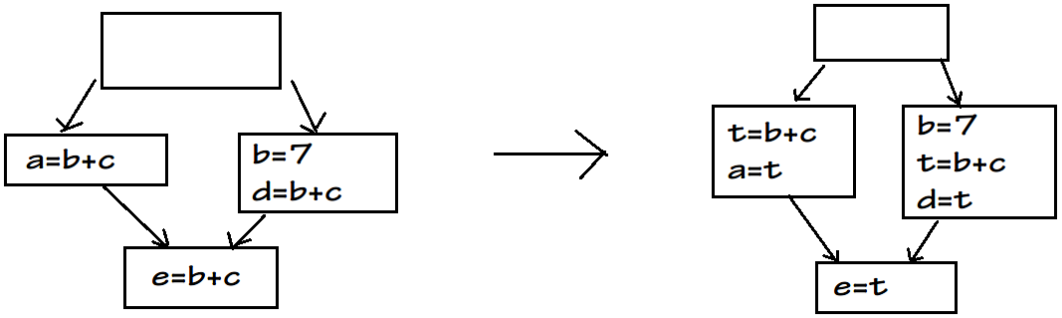
By assuming that the program is correct, the algorithm can find more constants than it otherwise would. That is, the algorithm conveniently chooses some values for those possibly undefined variables in order to make the program more efficient. This change is legal in most programming languages, since undefined variables are allowed to take on any value. If the language semantics requires that all undefined variables be given some specific value, then we must change our problem formulation accordingly. And if instead we are interested in finding possibly undefined variables in a program, we can formulate a different data-flow analysis to provide that result.

**4. Partial-Redundancy Elimination**

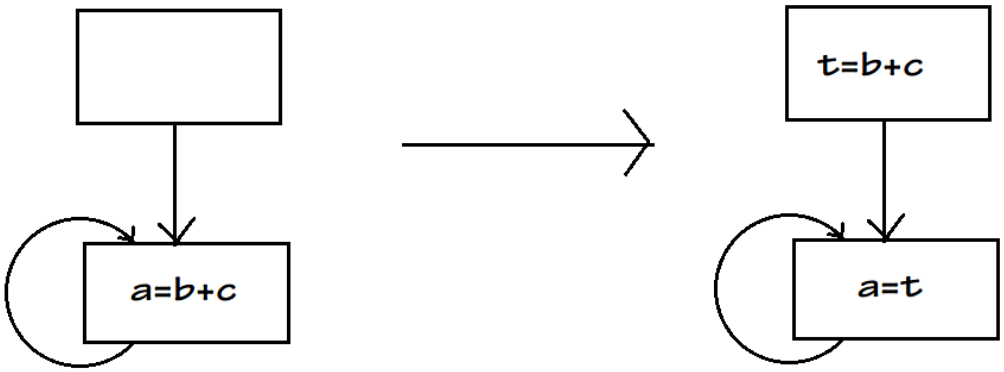
We want to consider all possible execution sequences in a flow graph, and look at the number of times that an expression such as $x + y$ is evaluated for. By moving around the places where $x + y$ is evaluated and keeping the result in a temporary variable when necessary, we often can reduce the number of evaluations of this expression along many of the execution paths, while not increasing that number along any path. Although the number of different places in the flow graph where $x + y$ is evaluated may increase, but that is relatively unimportant, as long as the number of evaluations of the expression $x + y$ is reduced.

There are forms of redundancy: common subexpressions, loop-invariant expressions, and partially redundant expressions. The following is some examples.

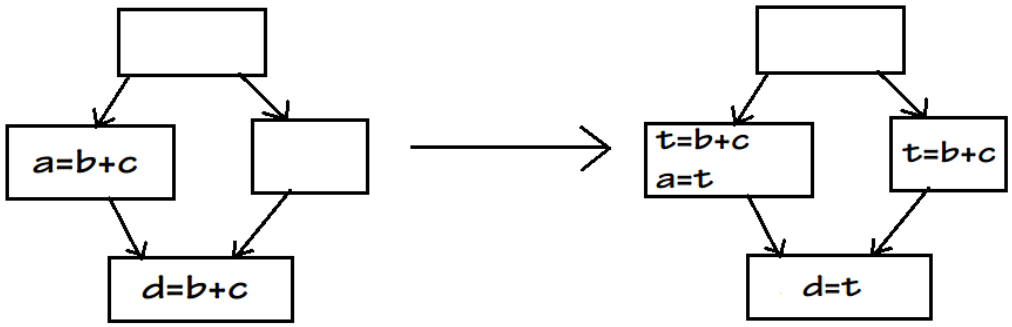
a) global common sub expression



b) loop-invariant code motion




c) partial-redundancy elimination




There is an additional constraint imposed on inserted expressions to ensure that no extra operations are executed. Copies of an expression must be placed only at program points where the expression is anticipated. We say that an expression $b + c$ is anticipated at point $p$ if all paths leading from the point $p$ eventually compute the value of the expression $b + c$ from the values of $b$ and $c$ that are available at that point.

We have a choice of where to place the copies of the expression, since there are usually several cut sets in the flow graph that satisfy all the requirements.

In the above, computation is introduced along the incoming edges to the path of interest and so the expression is computed as close to the use as possible, without introducing redundancy. These introduced operations may themselves be partially redundant with other instances of the same expression in the program. Such partial redundancy may be eliminated by moving these computations further up.

Anticipation of expressions limits how early an expression can be placed. One cannot place an expression so early that it is not anticipated where it is placed. The earlier an expression is placed, the more redundancy can be removed, and among all solutions that eliminate the same redundancies, the one that computes the expressions the latest minimizes the lifetimes of the registers holding the values of the expressions involved.

### 5. Loops in Flow Graphs

loops ate important because programs spend most of their time executing them, and optimizations that improve the performance of loops can have a significant impact. Thus, it is essential that we identify loops and treat them specially.

Loops also affect the running time of program analyses. If a program does not contain any loops, we can obtain the answers to data-flow problems by making just one pass through the program. For example, a forward data-flow problem can be solved by visiting all the nodes once, in topological order.

About this problem there exist iterative algorithms that have a high converge speed.

### 6. Region-Based Analysis

In the iterative-analysis approach, we create transfer functions for basic blocks, then find the fixed-point solution by repeated passes over the blocks. Instead of creating transfer functions just for individual blocks, a region-based analysis finds transfer functions that summarize the execution of progressively larger regions of the program. Ultimately, transfer functions for entire procedures are constructed and then applied, to get the desired data-flow values directly.

While a data-flow framework using an iterative algorithm is specified by a semilattice of data-flow values and a family of transfer functions closed under composition, region-based analysis requires more elements. A region-based framework includes both a semilattice of data-flow values and a semilattice of transfer functions that must possess a meet operator, a composition operator, and a closure operator.

A region-based analysis is particularly useful for data-flow problems where paths that have cycles may change the data-flow values. The closure operator allows the effect of a loop to be summarized more effectively than does iterative analysis. The technique is also useful for inter procedural analysis, where transfer functions associated with a procedure call may be treated like the transfer functions associated with basic blocks.

## V. Thinking and Feelings

As the final part, there are some thinking about the compiler and some thoughts about the course in this semester.

### 1. About the Compiler

Although I have learned a lot about the compiler from the course, but there are still many knowledges waiting to be explored.

As a relative underlying tool in programming, the compiler takes an important role to support the programming producing. The efficiency of the compiler can directly influence the running performance of the program. Good compilers can help programmers a lot, own to their error report mechanism and optimization to code. Today there are many outstanding compilers like

gcc, clang, and tools to build a compiler like LLVM. These all make the programming environment better. And the improvement of the compiler can be still worked on in long time, especially code optimization.

## 2. About this Course

I have learned a lot from this course. Both the online teaching and the homework benefit me a lot. And the big project gives me a chance to finish a simple compiler. Although that is done with my groupmates' help, I have experienced many problems and gain a lot. Some pities are we haven’t added much optimization to our compiler and we have only implemented some basic grammar characteristics.

This report makes me to read some reference materials besides the textbook and I have learned more about relational fields during the writing.

Thanks a lot!