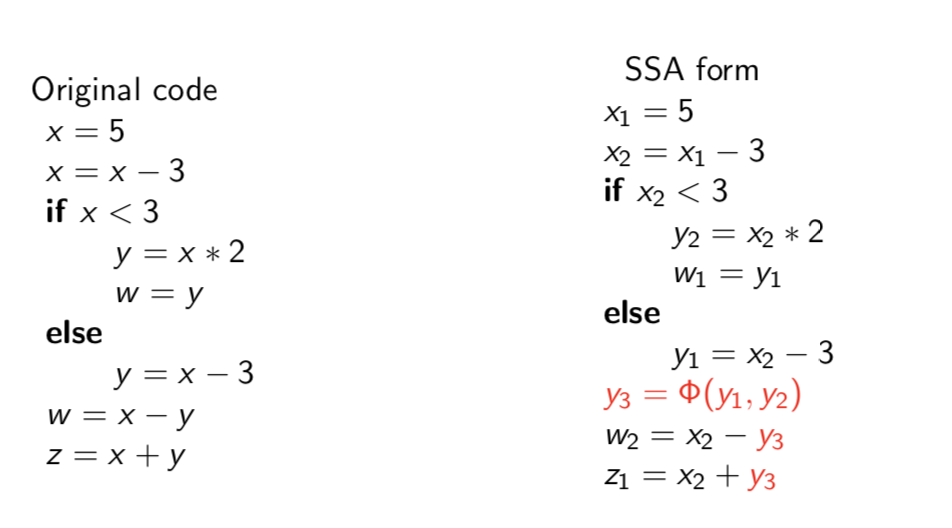
# Intermediate Code Generation

* http://www.montefiore.ulg.ac.be/~geurts/Cours/compil/2012/05-intermediatecode-2012-2013.pdf
  + Intermediate code generation is done in the final phase of the compiler front end following the semantic analysis phase
  + Goal: translate the program into a format that is expected by the compiler backend
  + In most compilers this is followed by intermediate code optimization and machine code generation
  + Why is this used?
    - Easy to change source or target language by adapting only the front end or back end
    - Makes it easier to write optimization methods for only the intermediate representations
    - Can be directly interpreted
  + How to choose intermediate representations:
    - Easy to translate source language to intermediate representation
    - Easy to translate intermediate representation to the machine code
    - Should be suitable for optimization
    - Should not be too high or low level
  + The Intermediate Language
    - Can use three address code which is similar to machine code. This is done by having a series of instructions written out
    - Values are all assumed to be ints
    - Unary and binary operators include normal arithmetic and logical operations
    - Atomic expressions are either a variable or a constant
    - Labels mark the position in a program
  + Principle of Translation:
    - Syntax-directed translation using several attributes:
      * Code returned as synthesized attribute
      * Symbol tables passed as inherited attributes
      * Places to store intermediate values as synthesized or inherited attributes
    - Implementing recursive functions defined on syntax tree nodes
    - Translation follows syntax = suboptimal code 🡪 optimization is next phase
    - Expressions:
      * Every operation is stored into a new variable in the intermediate language and is generated by a newvar function
      * Variables for sub-expressions are created by parent expressions and passed to a sub-expression as inherited attributes
      * Getopnames will retrieve operator associated with the toke unop, and transop translates this operator into the equivalent operator in the intermediate code.
    - Statements:
      * Unused new labels are generated by a label function, which are created by their parents and passed as inherited attributes
      * Sequences are dealt with by completing them in the order of the code, and then placing them together
      * Conditionals statements, such as if statements are done by dealing with the first statement, then using a compare and a GoTo label to go to the else conditions
      * While loops are done by dealing first with the condition, and then the statement, and using goTos to deal with breaking out of the loop
      * Logical operators can be done in two ways. One by arguments evaluated then the operators applied, or through sequential logical operators where the second operand is not evaluated if the first can determine the result in itself
        + The first can be applied by allowing any expression as a condition and including ‘&’, ‘|’, and ‘!’ among binary and unary operators
        + The second will require more modifications
      * Gotos and labels: labels are stored in the symbol table and associated with the intermediate language labels, and can be generated as soon as a jump or break is met
      * Break/exit: pass an additional inherited attribute to the translation function of loops with the label a break/exit should jump to, and a new label is generated when entering a new loop
      * Switch/case statements: translated with nested if-then-else statements
      * Arrays: can be allocated either statically at compile time or dynamically at run-time. Base address of the array is stored as a constant in the state of static allocation or in a variable in the case of dynamic allocation. The symbol table will bind the name of the constant or variable containing its address
      * Multi-dimensional arrays: two ways to represent a 2d array in linear memory: Row-major: one row at a time or Column major: one column at a time
        + generalization if d0…dn-1 are the sizes of the dimensions in a n-dimensional array, Row-major: base +((...(i0 ·dim1 +i1)·dim2 ...+in-2)·dimn-1 +in-1)·size or Column-major: base +((...(in-1 ·dim0 +in-2)·dim1 ...+i1)·dimn-2 +i0)·size
        + Dimension sizes are stored as constant (static), in variables or in memory next to the array data (dynamic)
      * Floating point values: can be treated the same way as integers
      * Records/structures: allocated in a similar way as arrays, where each field is accessed by adding an offset to the base-address of the record. Base addresses and offsets for each field are stored in the symbol table for all record-variables
      * Strings: done similarly to arrays of bytes but with a length that can vary at run time
    - Variable declaration: information about where to find scalar variables like integers and arrays after declaration are stored in the symbol table
      * Allocations can be done in may ways/places 🡪 static, dynamic, local and global
      * This assumes that scalar variables are stored in the intermediate language variables and arrays are dynamically allocated on the heap with their base-addresses stored in a variable
    - Additional things:
      * Also need error checking for things such as array out of bounds, wrong number of dimensions, memory/heap overflow
      * Translation results are not returned in a string but as output into an array or file, or stored onto a structure such as a translation tree or a linked list, which the latter will allow for subsequent code restructuring during the optimization phase

# Three Address Code

* Three-address code is a linearized representation of the DAG, where interior nodes are named:
* Three-address code can be used as an intermediate representation between the front-end and back-end. Besides assignments with
* unary operators (e.g t1 := -a) and binary operators, copy instructions (e.g. a := t1), three-address code has:
  + Procedure call with parameters "param x" , "call p" ,"x:= call p"
* https://cs.nyu.edu/courses/spring11/G22.2130-001/lecture9.pdf
  + Another option for intermediate code presentation, based on the two concepts of addresses and instructions, with ad most one operator
  + Can be one of the following name of the source program, a constant, or compiler generated temporary
  + Instructions
    - Assignment and instructions are in the form x = y op z
    - Assignments of the form x = op y, copy instructions in the form of x = y
    - unconditional jumps for goto L,
    - conditional jumps of the form if x goto L and if false x goto L
    - conditional jumps such as if x relop y goto L
    - indexed copy instructions of the form x = y[i] and x[i] = y
    - addresses and pointer assignments of the form x = &y, x = \*y, and \*x = y

# SSA – Single-Static Assignment

* <http://www.montefiore.ulg.ac.be/~geurts/Cours/compil/2012/05-intermediatecode-2012-2013.pdf>
  + A naming discipline used to explicitly encode information about both the flow of control and the flow of data values
  + A program is in SSA form if: 1. each definition has a distinct name 2. each use refers to a single definition
  + Type and Declarations:
    - Need to check the types of the operands to make sure they are what is expected by the operator then determine the storage needed
    - Calculate the address of an array reference
    - Insert explicit type conversion and choose the right version of the operator
  + Storage Layout:
    - Multibyte objects are stored in consecutive bytes and given the address of the first byte
    - Storage for aggregates eg arrays and classes is allocated in one contiguous block of bytes
  + Main interest: allows to implement several code optimizations.
  + In the example above, it is clear from the SSA form that the first assignment is not necessary.
  + This can be done by adding something called  Φ-functions , in which it can be used in an ifstatement:
    - Φ(y1,y2) would be after the if statement and would be y1 if arrive through the then branch and y2 if through the else branch. They show up at all the places in the program where branches are merged
    - find a better pic
  + Benefits
    - Simultaneously simplifies and improves the results of a variety of compiler optimizations by simplifying the property variables
  + Converting to SSA form can be done by replacing the target of each assignment with a new variable and replacing those with the version of the variable that will reach that point

# JVM .class – Java Virtual Machine

# CLR

# LLVM – Low Level Virtual Machine