

# Chapter 10 Intermediate Representations

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: May 12, 2010

current version: January 25, 2011

Copyright ©January 25, 2011 by Wuu Yang. All rights reserved.

## Chapter outline: Intermediate Representations

1. Overview
2. Java virtual machine
3. Static single assignment form

## §10.1 Overview

The parser builds a *blank* AST. Semantics analysis adds semantic information to AST. Before generating the actual machine code, the compiler first generates a form of code known as *intermediate representation* (IR). IR is more concise and abstract and easier to generate than real machine code. Most compilers use one or more levels of IR before generating the real machine code.

We usually program in a high-level language, such as C or Java. High-level programs cannot be directly executed on hardware. Instead, only low-level instructions may be executed on hardware. A compiler bridges the high-level programs and the low-level instructions.

## §10.1.1 Examples

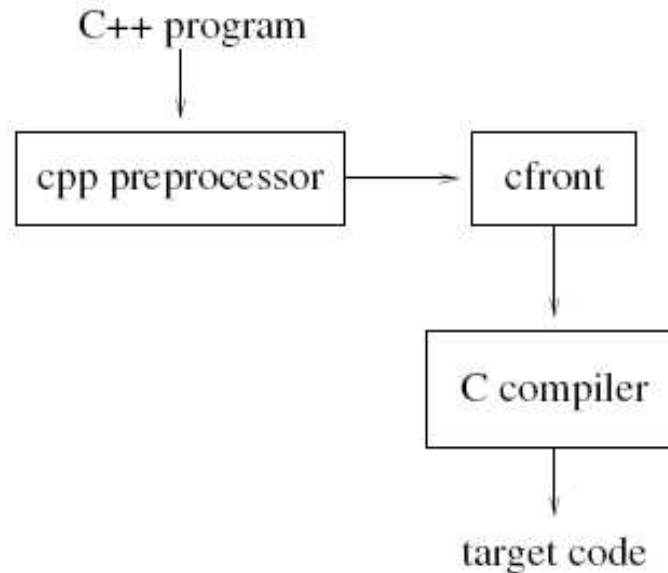


Figure 10.1: Use of *cfront* to translate C++ to C.

---

Early C++ uses *cpp* and *cfront* to translate C++ code to C code. The output from *cpp* and *cfront*, respectively, may be considered as an intermediate representation.

The *latex* formatting system, shown in Figure 10.2, also makes use of several intermediate languages: tex, dvi, and ps. These intermediate languages improve the portability of the *latex* system as new printers can be easily accommodated.

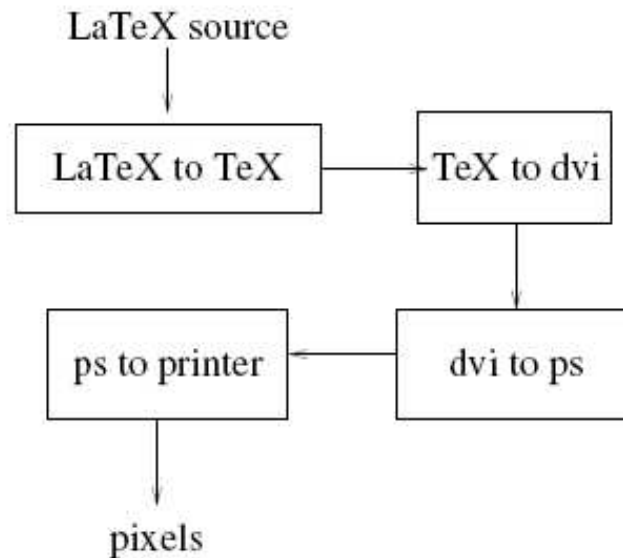


Figure 10.2: Translation from LaTeX into print.

---

The requirements of an intermediate language includes

1. An intermediate language must be precisely defined. That is, it must capture all the information in a source program.
2. There could be multiple intermediate languages. There are translators among the source language, intermediate languages, and the target language.
3. On each line in the intermediate representation, it should be possible to know the corresponding position in the source program. This makes, say, error messages easier to understand for a programmer.

Introducing several levels of intermediate representations in the compilation process actually slows down the process. However, appropriate IR's make it possible to break down the complicated compilation process into smaller steps and hence makes the task easier.

## §10.1.2 The middle-end

The *front-end* is responsible for parsing the input, which is more concerned with the source language.

The *back-end* is responsible for generating the target code, which is more concerned with the target platform.

In a compiler, there are usually a set of components between the front-end and the back-end, which is called the *middle-end*.

For a multiple-source, multiple-target compiler system, such as GCC, putting the components common to all compilers makes compiler development more economical. See Figure 10.3.

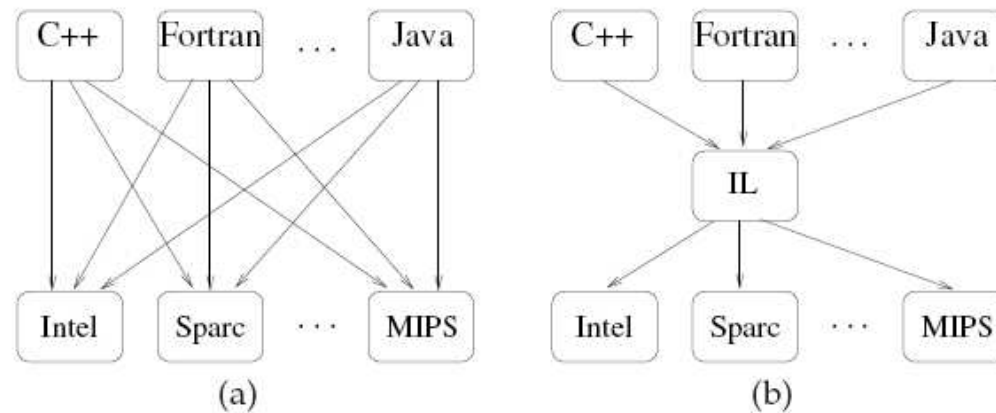


Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

Other advantages:

1. An IL allows many components to cooperate by sharing a common representation of programs. Information about a program, such as variable names, types, source lines, etc., is available. Many other tools, such as class browsers, debuggers, and profilers, can also be built around IR.

2. An IL simplifies development and testing.
3. A major portion of the IL can be re-used.
4. Different development teams/vendors can cooperate by agreeing on a common IL.
5. IL makes prototyping new algorithms (e.g., for CSE) and new compiler organizations (e.g., order of phases) easier in a research environment
6. The IL and its interpreter can serve as a reference definition of a programming language. For example, DIANA (descriptive intermediate attributed notation for Ada) is an IL that defines Ada.
7. Interpreters for an IL help testing and porting a compiler.
8. IL helps the development of retargetable code generators and hence enhances the portability of a compiler.

Common IL includes P-code (Pascal), bytecode (Java), bitcode (LLVM), DIANA (Ada), etc. GCC uses two ILs. MS C uses CIL (common intermediate language). CLR (common language runtime) is an interpreter for CIL.

## §10.2 Java virtual machine

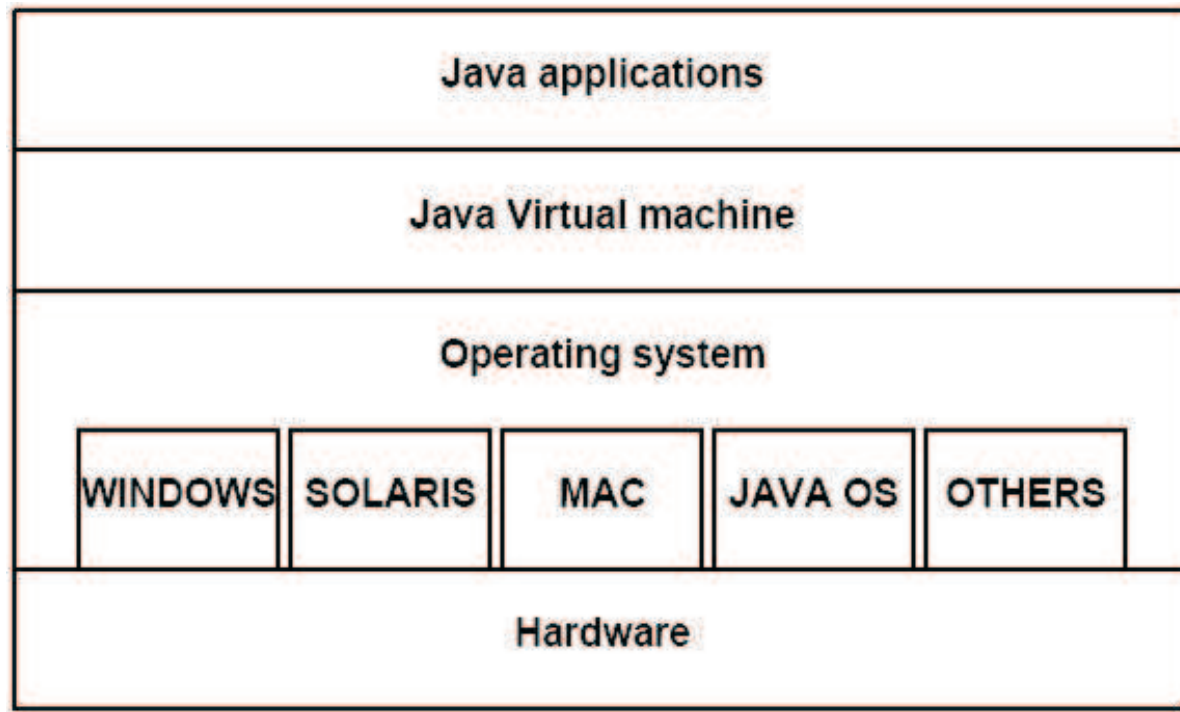


Figure 1: Java VM and applications

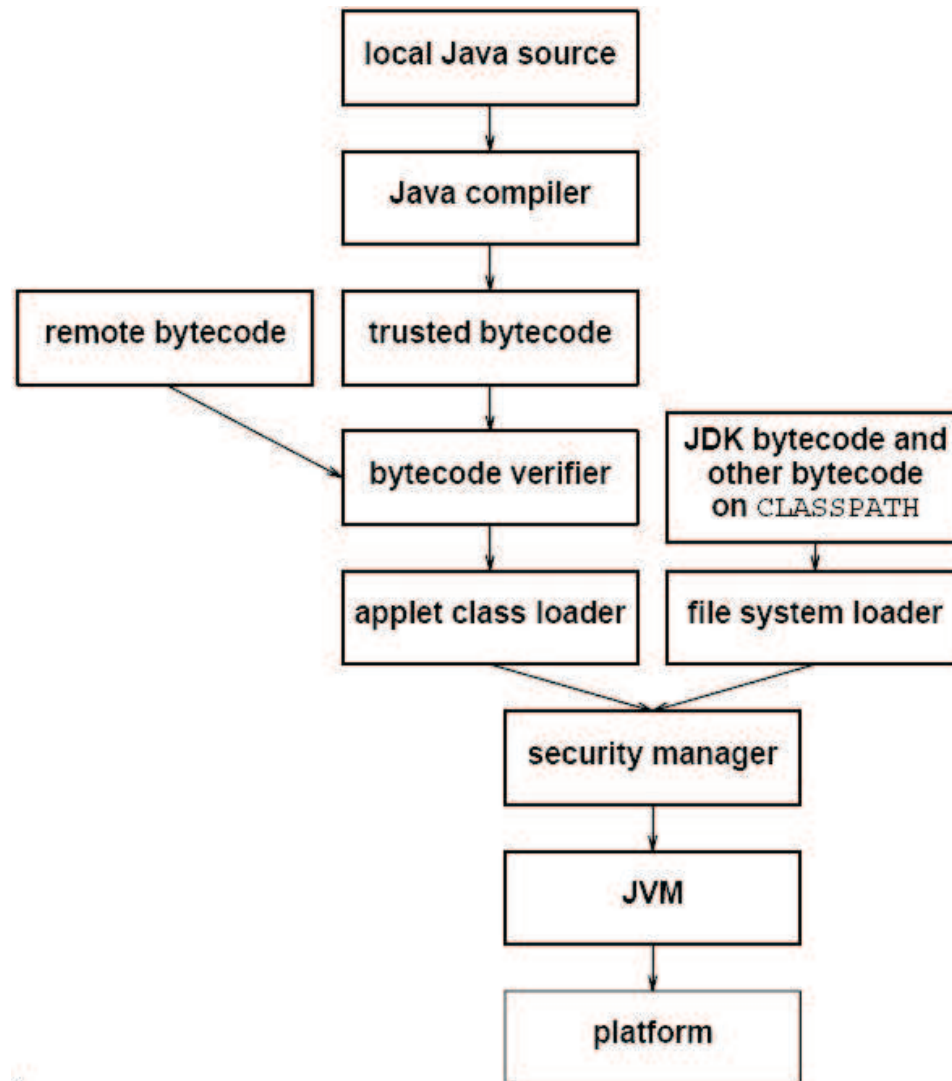


Figure 2: Java system

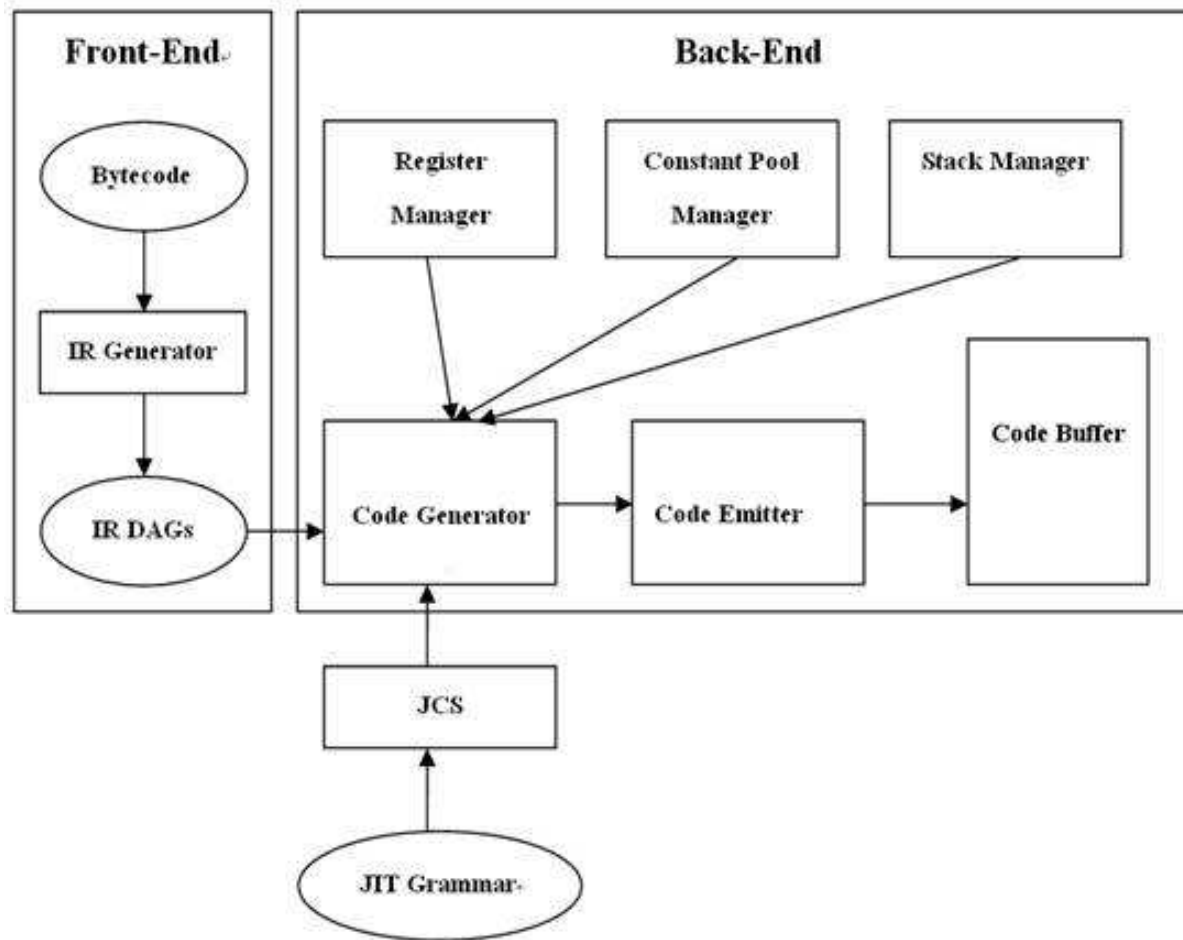


Figure 3: Structure of a JIT.

**Compactness** A Java compiler compiles Java source code to *bytecode*, which is similar to conventional `.exe` files. Bytecode is put in a *classfile* (`.class`).

Bytecode is compact. It is aimed at a stack machine (Java virtual machine). All instructions, such as `iadd`, operate on operands on the stack top. Thus, an instruction usually does not have operand addresses (and is short).

There is no register in the bytecode. Note that stack manipulation is slower than registers. The stack machine allows many short instructions since instructions have implicit operands on the stack. This leads to smaller code size. On the other hand, since stack manipulation is slower, a stack machine is usually slower than a register machine.

Sometimes there are several instructions (of different lengths) that achieve the same effect. For example, `iconst_0` (1 byte) and `ldc_w`

0 (3 bytes). Frequently used instructions have special short formats.

**Safety** JVM is designed to execute safely, not to compromise a user's computer. The instruction set and the virtual machine are designed with security in mind so that efficient security checks are possible. When a class file is loaded into JVM, a bytecode verifier will check the class file against various security requirements before execution.

In a purely zero-address form, to load a value of a register into the stack takes two steps:

1. First, compute the register number, and push it onto the stack
2. Next a `load` instruction pops off the register number and load the value of that register onto stack

Checking the validity of the register number must be done at run time.

In contrast, Java uses *immediate operand* (such as `iload 5`) as a register number. When loading the bytecode, the verifier can check

if register 5 is accessible by the current method. No run-time check is needed.

## §10.2. Contents of a class file

A class file is partitioned into several sections.

Java provides primitive and reference types. Figure 10.4 shows the abbreviations for types. Below is an example reference type:

```
Ljava/lang/String;
```

Type	JVM designation
boolean	Z
byte	B
double	D
float	F
int	I
long	J
short	S
void	V
Reference type <i>t</i>	L <i>t</i> ;
Array of type <i>a</i>	[ <i>a</i> ]

Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

---

A class file contains a *constant pool*, which contains all the constants (integer, real, strings, etc.) used in that method. A constant is referenced through its ordinal position (1, 2, etc.), not by its byte-offset, inside the constant pool.

Some instructions, such as `ldc`, uses 1 byte for constant-pool references. Some instructions, such as `ldc_w`, uses 2 bytes for constant-pool references. (Hence, `ldc_w` can reference more entries.)

### §10.2.3 JVM instructions

JVM supports the following kinds of instructions:

1. Arithmetic: `iadd` pops off two values from the stack, adds them as integers (32-bit 2's complement), and pushes the result back to the stack. There are other additions: `fadd`, `ladd`, `dadd`. There are similar instructions for subtraction, multiplication, division, and remainder.
2. Register traffic: JVM has no real registers. Instead, it has an unlimited number of *virtual registers*. The virtual registers are used for a method's local variables. For static methods, register 0 is the 1st argument. For instance methods, register 0 holds `this`. When a method is invoked, argument values are automatically popped from the caller's stack and deposited into the low-numbered registers. JVM registers are untyped. `long` and `double` occupy a pair of even-odd registers. An example,

`iload 2` takes two bytes. A similar instruction `iload_2` takes one byte.

Example: `istore 10`.

Example: `fload n`.

JVM uses an integer interpretation for boolean: 0 as false and 1 as true.

`char`, `byte`, and `short` are treated as `int`.

An object reference takes four bytes. It is loaded and stored with `aload 4` and `astore 5` instructions. 0 is treated as `null`.

The bytecode verifier will check for type errors, such as “`astore 4; iload 4`”.

3. Registers and types: The bytecode verifier checks the types of all values (in registers and stack). “`iload 4; fload 5`” will cause a type error before execution. Instead, we should use “`iload 4; i2f, fload 5`”.

4. Static fields are manipulated with `getstatic` `fieldname` `type` and `putstatic` instructions. Example:

```
getstatic java/lang/System.out Ljava/io/PrintStream
```

The `getstatic` `fieldname` `type` instruction takes 3 bytes: one for the operator and the remaining two are an entry in the constant pool. That entry will contain the `fieldname` and `type`.

(A compiler will generate the correct `type` in a `getstatic` field. What if a hacker modifies the `type`? How and will the bytecode verifier check this?)

`putstatic` pops a value off the stack and stores it into the specified location.

5. Instance fields are similarly manipulated with `getfield` and `putfield` instructions. “`getfield x I`” needs to use the entry on the stack top, which is an object reference.

“puttfield x I” stores the value on stack top to the specified field. It also needs to use the entry on the stack top, which is an object reference.

6. Branching: `goto delta` takes three bytes. The last two bytes forms a 16-bit offset from the location of the `goto` instruction. There is also a 5-byte version: `goto_w delta`, which accommodates a 4-byte offset.

There are also conditional jump instructions. Some conditional jumps combine comparison with jump. Others make use of two instructions: one for comparison and the other for conditional jump. Examples: `if_icmpeq` (compare two values on stack top), `if_icmpne`, `if_icmplt`, `if_icmple`, `if_icmtgt`, `if_icmtge`, `ifeq` (compare the number on stack top with zero), `ifne`, `iflt`, `ifle`, `ifgt`, `ifge`.

7. Calling a static method:

```
invokestatic java/lang/Math/pow(DD)D
```

pops two double values from the stack, calculates the exponentiation, and pushes the result back on stack top. The parameters are pushed from left to right. The method name and signature are stored in the constant pool. The `invokestatic` instruction uses an ordinal index into the constant pool.

8. Calling an instance method:

```
invokevirtual java/io/PrintStream/print(Z)V
```

The parameters and the implicit object reference are taken from the stack.

9. All constructors are named `<init>`, which are invoked with the `invokespecial` instruction.

10. stack operations: `dup` makes a copy of the value on stack top. `new t` leaves a reference on TOS. Then a constructor will be

invoked to use that reference. A `dup` instruction is used to duplicate that reference. Note the bizarre behavior of `dup_x1`.

### **§10.3 Static single assignment form**

Static single assignment form makes a program's data flow (def-use relation) explicit from the names of variables.

<pre> <i>i</i> ← 1 <i>j</i> ← 1 <i>k</i> ← 1 <i>l</i> ← 1 repeat     if <i>p</i>     then         <i>j</i> ← <i>i</i>         if <i>q</i>         then <i>l</i> ← 2         else <i>l</i> ← 3          <i>k</i> ← <i>k</i> + 1     else <i>k</i> ← <i>k</i> + 2      call    (<i>i</i>, <i>j</i>, <i>k</i>, <i>l</i>)     repeat         if <i>r</i>         then             <i>l</i> ← <i>l</i> + 4      until <i>s</i>     <i>i</i> ← <i>i</i> + 6 until <i>t</i> (a) </pre>	<pre> <i>i</i><sub>1</sub> ← 1 <i>j</i><sub>1</sub> ← 1 <i>k</i><sub>1</sub> ← 1 <i>l</i><sub>1</sub> ← 1 repeat     <i>i</i><sub>2</sub> ← φ(<i>i</i><sub>3</sub>, <i>i</i><sub>1</sub>)     <i>j</i><sub>2</sub> ← φ(<i>j</i><sub>4</sub>, <i>j</i><sub>1</sub>)     <i>k</i><sub>2</sub> ← φ(<i>k</i><sub>5</sub>, <i>k</i><sub>1</sub>)     <i>l</i><sub>2</sub> ← φ(<i>l</i><sub>9</sub>, <i>l</i><sub>1</sub>)     if <i>p</i>     then         <i>j</i><sub>3</sub> ← <i>i</i><sub>2</sub>         if <i>q</i>         then <i>l</i><sub>3</sub> ← 2         else <i>l</i><sub>4</sub> ← 3         <i>l</i><sub>5</sub> ← φ(<i>l</i><sub>3</sub>, <i>l</i><sub>4</sub>)         <i>k</i><sub>3</sub> ← <i>k</i><sub>2</sub> + 1     else <i>k</i><sub>4</sub> ← <i>k</i><sub>2</sub> + 2     <i>j</i><sub>4</sub> ← φ(<i>j</i><sub>3</sub>, <i>j</i><sub>2</sub>)     <i>k</i><sub>5</sub> ← φ(<i>k</i><sub>3</sub>, <i>k</i><sub>4</sub>)     <i>l</i><sub>6</sub> ← φ(<i>l</i><sub>2</sub>, <i>l</i><sub>5</sub>)     call    (<i>i</i><sub>2</sub>, <i>j</i><sub>4</sub>, <i>k</i><sub>5</sub>, <i>l</i><sub>6</sub>)     repeat         <i>l</i><sub>7</sub> ← φ(<i>l</i><sub>9</sub>, <i>l</i><sub>6</sub>)         if <i>r</i>         then             <i>l</i><sub>8</sub> ← <i>l</i><sub>7</sub> + 4         <i>l</i><sub>9</sub> ← φ(<i>l</i><sub>8</sub>, <i>l</i><sub>7</sub>)      until <i>s</i>     <i>i</i><sub>3</sub> ← <i>i</i><sub>2</sub> + 6 until <i>t</i> (b) </pre>
---	---

Figure 10.5: SSA Form example taken from [CFR+91]. Program (b) shows the SSA Form for program (a).

## Applications of SSA:

1. SSA based register allocation, see “SSA-register-allocation.pdf”.