UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Intermediate Representaions

Concepts of Programming Languages (CoPL)

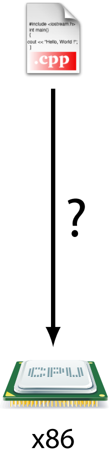
Malte Skambath
malte@skambath.de

November 16, 2015

Overview

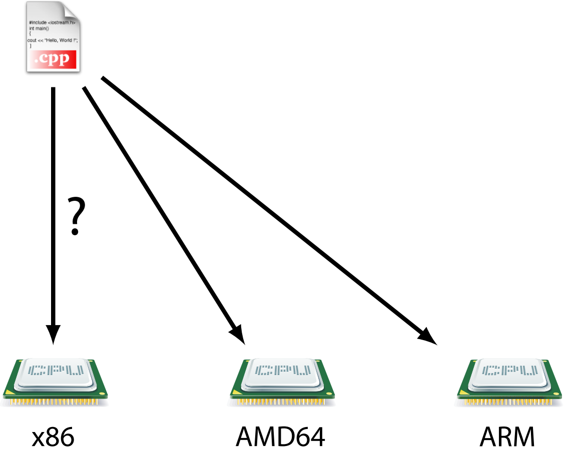
Developing Software

We need compilers!



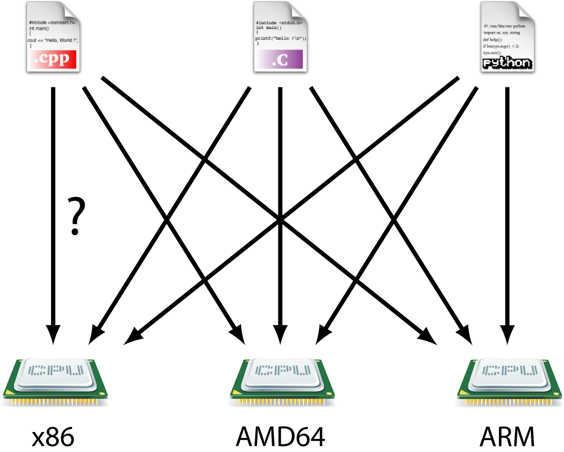
Developing Software

We need compilers!



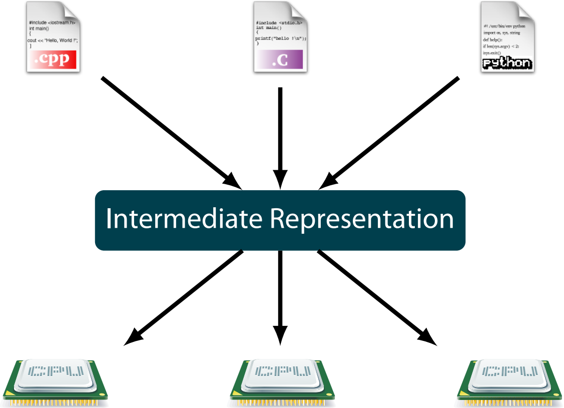
Developing Software

We need compilers!



Intermediate Representation

The solution!



Intermediate Representation

x86

AMD64

ARM

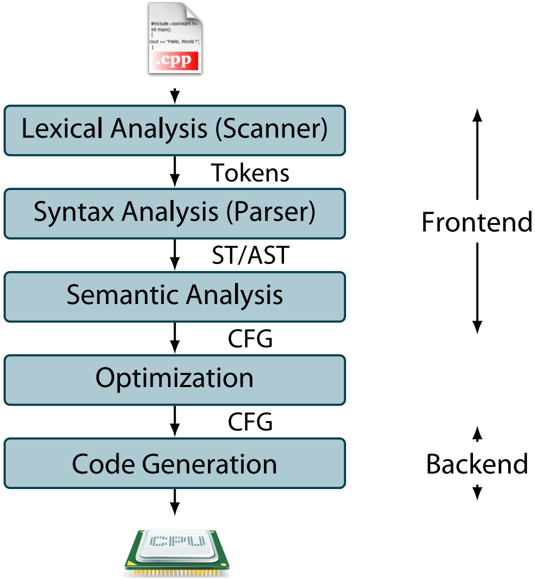
Intermediate Representation

Definition

An *intermediate representation* (IR) is data structure as representation of a program between a high-level programming language and machine code.

An *intermediate language* (IL) is a low-level assembly language as IR for a virtual machine.

Classical Compiler Process



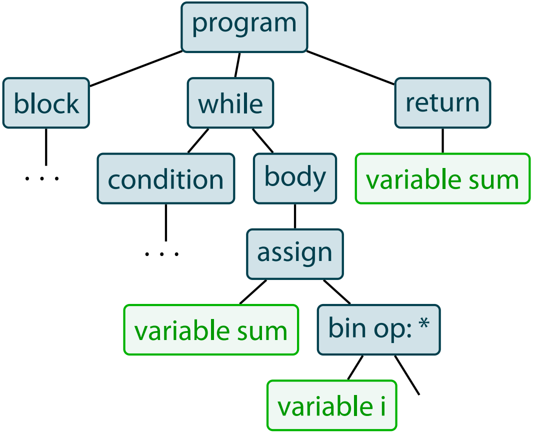
Abstract Sytax Tree

An *abstract syntax tree (AST)* . . .

. . . describes the syntactical structure of a program

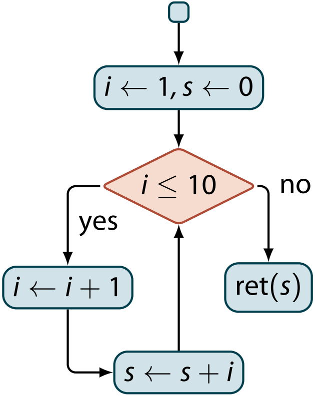
. . . depends on the programming language

. . . is generated during by the parser



Control-Flow-Graph

```
int s = 1;
for(int i=1; i<=10; i++)
  s += i;
return (s);
```



Stack Machines

Definition

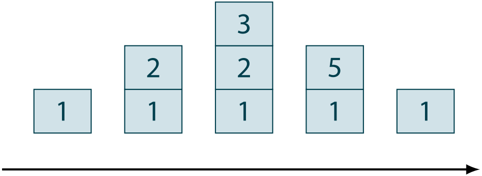
A general *Stack Machine* has

- a stack as storage
- a set of instructions / operations $\mathrm{op} = F(a_1, a_2, \ldots, a_n)$ including (`push` and `pop`)

Executing an operation takes the arguments from top of the stack, computes the result in the accumulator, and pushes the result back the stack.

Example

```
push 1
push 2
push 3
add
pop
```



Stack-machines

Code Generation

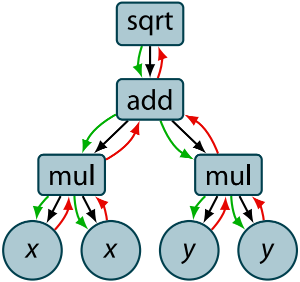
We can generate the control by traversing the syntax tree.
Assume we have to compute the expression $\sqrt{x^2 + y^2}$.

Stack-machines

Code Generation

We can generate the control by traversing the syntax tree. Assume we have to compute the expression $\sqrt{x^2 + y^2}$.

AST



```
push x
push x
mul
push y
push y
mul
add
sqrt
```

Stack Machines

Summary

- Programs for stack machines are short
  Only the opcodes (or constants) in the byte code.
- In practical use stack machines can be extended
  1. An external memory to store and load values (computations are still limited to the stack)
  2. Top-Level registers
  3. Metainformations (see CIL later)
- Problem: Most processor-architectures use registers.
  $\Rightarrow$ Hybrid Models, Special informations in the intermediate representation.

Register Machines

Definition

A *register machine* . . .

- consists of an infinite number of memory cells named *registers*
- each register is accessible
- has a limited set of instruction / operations:
  1. Arithmetical Operations: Computes a function F using selected registers $\langle o_1, \rangle \ldots, \langle o_n \rangle$ as operands and stores the result in a target register $\langle r \rangle$
  2. Jumps/Branches

Three-Address Code (3AC/TAC)

- Each TAC is a sequence of instructions $I_1, I_2, \ldots, I_n$ for a register machine.
- Instructions can be
  1. Assignments `r1 := r0`
  2. Unconditional Jumps (Instructions can be labeled)

     ```
     L0: goto L1
     ...
     L1: r0 := 1
     ```

  3. Conditional Branches

     ```
     if a<b then goto L1
     ```

  4. Arithmetical operations `r3 := add(r1,r2)`
- Each instruction contains at most *3* registers

Three-Address Code (3AC/TAC)

- Each TAC is a sequence of instructions $I_1, I_2, \dots, I_n$ for a register machine.
- Instructions can be
  1. Assignments `r1 := r0`
  2. Unconditional Jumps (Instructions can be labeled)

```
L0: goto L1
...
L1: r0 := 1
```

  3. Conditional Branches

```
if a<b then goto L1
```

  4. Arithmetical operations `r3 := add(r1,r2)`
- Each instruction contains at most *3* registers

Example ($\sqrt{x^2 + y^2}$)

```
t1 :=  x * x
t2 :=  y * y
t3 := t1 + t2
result := sqrt(t3)
```

Three-Address Code (3AC/TAC)

How to design the Byte-Code

For practical use we should store TAC in byte code format.

- Each operation has an *opcode* for the virtual machine
- Each instruction can be represented by tuples

Quadruples

	opcode	op1	op2
t1	MUL	x	x
t2	MUL	y	y
t1	ADD	t1	t2
res	SQRT	t1	-

Triples

opcode	op1	op2
MUL	x	x
MUL	y	y
ADD	(1)	(2)
SQRT	(3)	-

Note

Registers can be assigned implicitly (Triples). But then each register has to be assigned only once.

Static-Single-Assignment

Definition (Static-Single Assignment)

A Three-Adress Code is in *Static-Single Assignment*-from if each register gets assigned once in the code.

Example ($\sqrt{x^2 + y^2}$)

Not in SSA

```
L1: x := x * x
L2: y := y * y
L3: x := x + y
L4: z := sqrt(x)
```

SSA

```
L1: x0 := x * x
L2: y0 := y * y
L3: x1 := x0 + y0
L4: z  := sqrt(x1)
```

Static-Single-Assigment

How to get SSA-form?

A simple Algorithm

- For each used register: $\langle R\rangle$
  1. Check if $\langle R\rangle$ gets assigned more than once
  2. For each assignment/definition of $\langle R\rangle$:
     - Rename on the left side to $\langle R.i\rangle$ if this assignment is the i-th assignment to $\langle R\rangle$
  3. For each use of $\langle R\rangle$:
     - Replace $\langle R\rangle$ with $\langle R.j\rangle$ where $\langle R.j\rangle$ was the previous replacement for $\langle R\rangle$.

Is this algorithm correct?

Static-Single-Assigment

How to get SSA-form?

A simple Algorithm

- For each used register: $\langle R \rangle$
  1. Check if $\langle R \rangle$ gets assigned more than once
  2. For each assignment/definition of $\langle R \rangle$:
     - Rename on the left side to $\langle R.i \rangle$ if this assignment is the i-th
       assignment to $\langle R \rangle$
  3. For each use of $\langle R \rangle$:
     - Replace $\langle R \rangle$ with $\langle R.j \rangle$ where $\langle R.j \rangle$ was the previous replacement for $\langle R \rangle$.

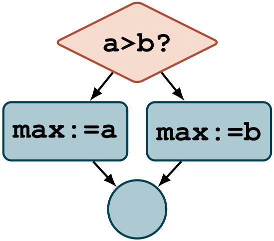
Is this algorithm correct?

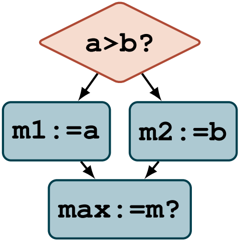
No!

Static-Single-Assignment

What if we have branches?

```
  if a>b then goto L_A
  max := b;
  goto L_END
L_A:
  max := a;
  goto L_END
L_END:
```

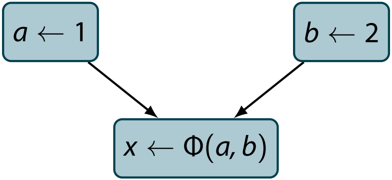




Static-Single-Assignment

The Φ-function

The Φ-*function* computes the value depending on the incoming branch.



$a \leftarrow 1$

$b \leftarrow 2$

$x \leftarrow \Phi(a, b)$

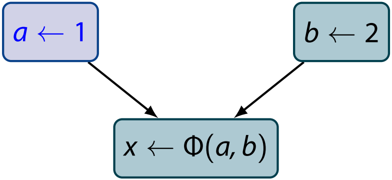
Note

There is no real operation like Φ in real machines. After optimization Φ-statements have to be removed.

Static-Single-Assignment

The Φ-function

The Φ-*function* computes the value depending on the incoming branch.



$a \leftarrow 1$

$b \leftarrow 2$

$x \leftarrow \Phi(a, b)$

x has value 1

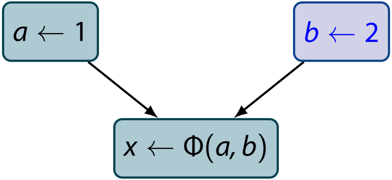
Note

There is no real operation like Φ in real machines. After optimization Φ-statements have to be removed.

Static-Single-Assignment

The Φ-function

The *Φ-function* computes the value depending on the incoming branch.



$a \leftarrow 1$

$b \leftarrow 2$

$x \leftarrow \Phi(a, b)$

x has value 2

Note

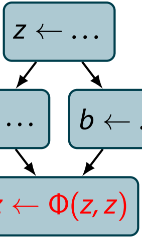
There is no real operation like Φ in real machines. After optimization Φ-statements have to be removed.

Getting Code in SSA-form.

```
L1:
if r_a < r_b then goto L3:
L2:
t_1 := r_a
goto L4

L3:
t_2 := r_b
goto L4

L4: max := phi t_1 [from L2], t_2 [from L3]
```

Converting to SSA-Form

1. Place Φ-function terms
2. Rename registers to achieve SSA-form



Using the Φ-function after each branch for previous registers is an unpractical solution.

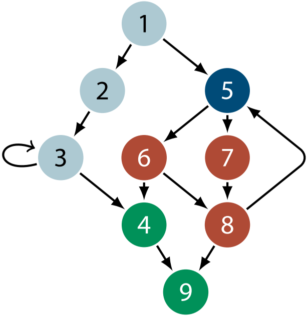
Dominance Frontiers

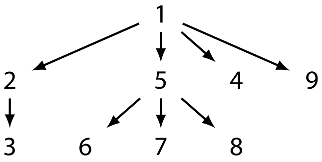
Definition

We say *x dominates y* (x dom y) if on all paths to Y in the CFG the program has to run over X.

Definition

y is in the *dominance frontier* of x (DF(x)) iff not x dom y and y has a direct predecessor on all paths to y

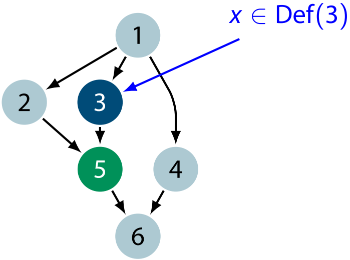




$$\mathrm{DOM}(5) = \{5, 6, 7, 8\}$$
$$\mathrm{DF}(5) = \{4, 9\}$$

Dominance Frontiers

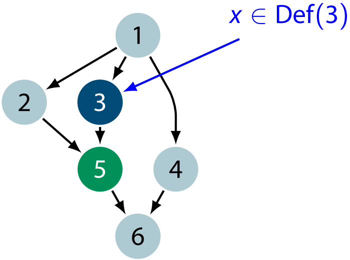
Assume that node 3 defines variable x, $\mathrm{DF}(3) = \{5\}$



Is 5 the only node we need to insert a Φ-function for x?

Dominance Frontiers

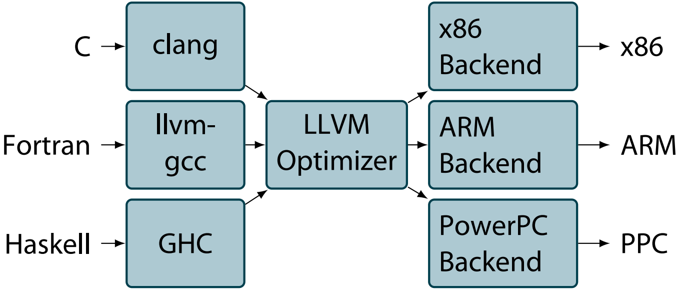
Assume that node 3 defines variable x, $\mathrm{DF}(3) = \{5\}$



Is 5 the only node we need to insert a Φ-function for x?

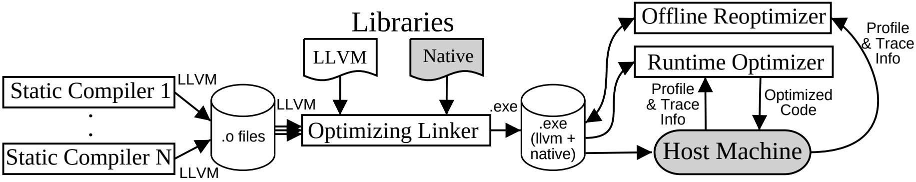
No, at node 6. Why?

Architecture of the LLVM-compiler process



LLVM uses a special intermediate representation (LLVM-IR) for a virtual register machine.

LLVM Compilation Strategy



C. Lattner, The LLVM Instruction Set and Compilation Strategy, 2002

LLVM-IR

```
@.str = private unnamed_addr constant [11 x i8] c"
    ␣%d␣<=␣%d␣\00", align 1

; Function Attrs: nounwind uwtable
define void @minmax(i32 %a, i32 %b) #0 {
  %1 = icmp sgt i32 %a, %b
  br i1 %1, label %2, label %3

  ; <label>:2                        ; preds = %0
  br label %4

  ; <label>:3                        ; preds = %0
  br label %4

  ; <label>:4                        ; preds = %3, %2
  %max.0 = phi i32 [ %a, %2 ], [ %b, %3 ]
  %min.0 = phi i32 [ %b, %2 ], [ %a, %3 ]
  %5 = call i32 (i8*, ...) @printf(i8*
    getelementptr inbounds ([11 x i8], [11 x i8]*
    @.str, i32 0, i32 0), i32 %min.0, i32 %max.0)
  ret void
}
```

LLVM-IR

- LLVM is register-based. Registers are written as `%<registername>` (e.g. `%R1 = ...`)
  `@` is used for global variables (e.g. function names)
- LLVM use types `i1, i8, i32` for boolean, Byte and 32-Bit Integer values
- Reduced instruction-set
  - Memory Access `%ptr = alloca i32`
  - Comparing `%res = icmp <opt> <type> %a, %b`
- Conditional Branches
  `br i1 %cond, label %IfLabel, label %ElseLabel`
- Function calls `%res = call`
- `phi`-Instruction for assignments depending on the control flow
- Functions:
  `define <type> @FctName(<type> %arg1,...){...}`
- Metadata

LLVM-IR

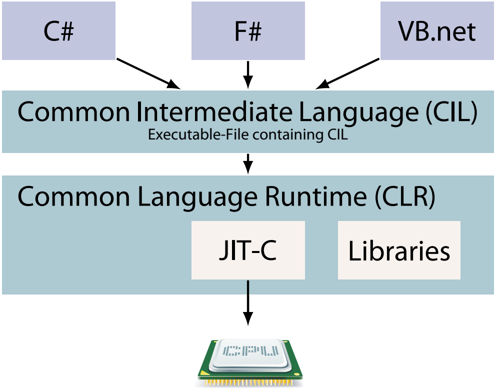
```
@.str = private unnamed_addr constant [11 x i8]
  c"␣%d␣<=␣%d␣\00", align 1

; Function Attrs: nounwind uwtable
define void @minmax(i32 %a, i32 %b) #0 {
%1 = icmp sgt i32 %a, %b
br i1 %1, label %2, label %3

; <label>:2                           ; preds = %0
br label %4

; <label>:3                           ; preds = %0
br label %4

; <label>:4                           ; preds = %3, %2
%max.0 = phi i32 [ %a, %2 ], [ %b, %3 ]
%min.0 = phi i32 [ %b, %2 ], [ %a, %3 ]
%5 = call i32 (i8*, ...) @printf(i8*
  getelementptr inbounds ([11 x i8], [11 x i8]*
  @.str, i32 0, i32 0), i32 %min.0, i32 %max.0)
ret void
}
```

LLVM-IR

Another example

```
define i32 @main() #0 {
  %c = alloca [10 x i32], align 16
  br label %1
1:
  %sum.0 = phi i32 [ 0, %0 ], [ %4, %7 ]
  %i.0   = phi i32 [ 1, %0 ], [ %8, %7 ]
  %2 = icmp sle i32 %i.0, 10
  br i1 %2, label %3, label %9
3:                                    ; preds = %1
  %4 = add nsw i32 %sum.0, %i.0
  %5 = sext i32 %i.0 to i64
  %6 = getelementptr inbounds [10 x i32], [10 x
    i32]* %c, i32 0, i64 %5
  store i32 %4, i32* %6, align 4
  br label %7
7:                                    ; preds = %3
  %8 = add nsw i32 %i.0, 1
  br label %1
9:                                    ; preds = %1
  ret i32 0
}
```

Common Intermediate Language



C#

F#

VB.net

Common Intermediate Language (CIL)

Executable-File containing CIL

Common Language Runtime (CLR)

JIT-C

Libraries

Common Language Infrastructure (CLI)

The CLR is the CLI-Implementation of Microsoft and part of the .net-Framework. The CLI also speciefies a Type System (CTS) and a basic set of class libraries.

CIL

- Stack based virtual machine.
- Each method has a header
- Typed instruction-set (e. g. `ldc.i4.0` load constant 0 as 4-Byte int)
- Access to local variables `ldloc.<index>`, `stloc.<index>`
- Object oriented
  - Load field values `ldfld string Program/Person::prename`
  - Create new objects `newobj instance void class <CLASS>'.ctor'(...)`

CIL

An Example

```
.method public static hidebysig
default int32 sum (int32 a, int32 b)  cil
managed {
.maxstack 2
.locals init (int32 V_0, int32  V_1)
IL_0000:  ldc.i4.0    //
...
}
```

CIL

An Example

```
IL_0000:  ldc.i4.0     //
IL_0001:  stloc.0      // sum = 0
IL_0002:  ldarg.0      // load a on the stack
IL_0003:  stloc.1      // store a in first var
(i=a)
IL_0004:  br IL_0011   // --+
IL_0009:  ldloc.0      //   |    <--+
IL_000a:  ldloc.1      //   |       |
IL_000b:  add          //   |
IL_000c:  stloc.0      //   |
IL_000d:  ldloc.1      //   |
IL_000e:  ldc.i4.1     //   |
IL_000f:  add          //   |       .
IL_0010:  stloc.1      //   |       .
IL_0011:  ldloc.1      // <-+       .
IL_0012:  ldarg.1      // load b    |
IL_0013:  ble IL_0009  // i<=b     -+
IL_0018:  ldloc.0
IL_0019:  ret
```

Conclusion

Intermediate Representations . . .

- allow a clean and general compiler-architecture/infrastructure
- allow mixing different programming languages
- programmer may loose control on the real control-flow
- program-flow can be optimized
- adaption to different hardware configurations (including GPU-support).
- improve the development of new programming languages
- can realize translations between different languages

Any Questions?