**Script** generated by TTT

Title: Seidl: Virtual_Machines (11.04.2016)

Date: Mon Apr 11 10:08:59 CEST 2016

Duration: 82:14 min

Pages: 42

Helmut Seidl

# Virtual Machines

*München*

Summer 2016



## 0 Introduction

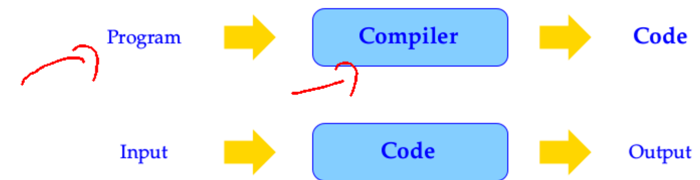
### Principle of Interpretation:

**Advantage:** No precomputation on the program text ⟹ no/short startup-time

**Disadvantages:** Program parts are repeatedly analyzed during execution + less efficient access to program variables
⟹ slower execution speed



### Principle of Compilation:




Two Phases (at two different Times):

- Translation of the source program into a machine program (at compile time);
- Execution of the machine program on input data (at run time).

Preprocessing of the source program provides for

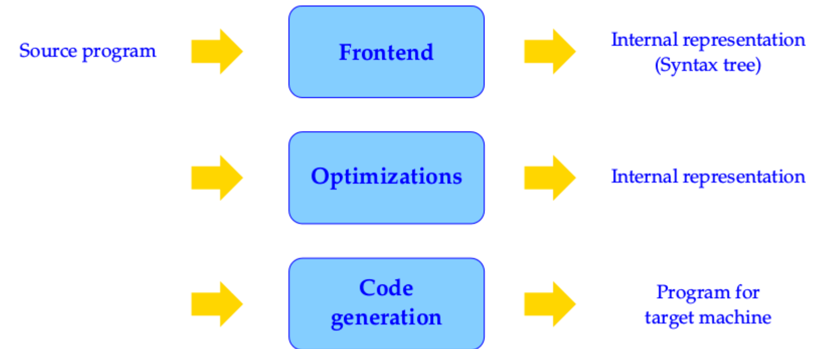
- efficient access to the values of program variables at run time
- global program transformations to increase execution speed.

**Disadvantage:** Compilation takes time

**Advantage:** Program execution is sped up $\Longrightarrow$ compilation pays off in long running or often run programs



## Structure of a compiler:





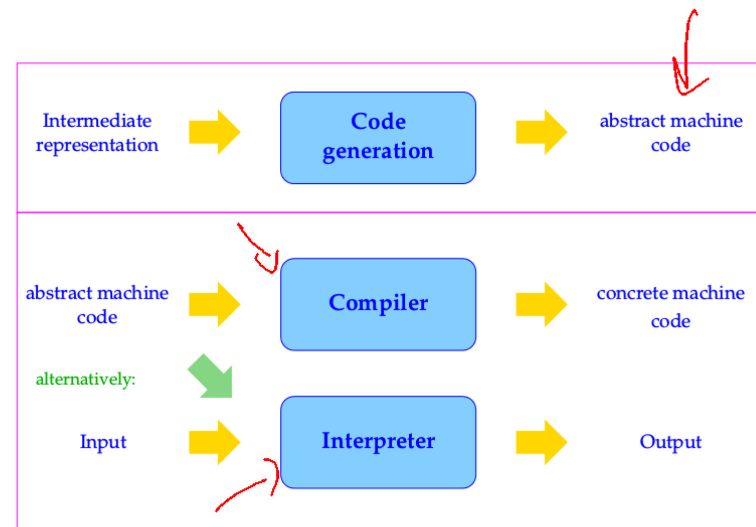

## Subtasks in code generation:

Goal is a good exploitation of the hardware resources:

1. Instruction Selection: Selection of efficient, semantically equivalent instruction sequences;
2. Register-allocation: Best use of the available processor registers
3. Instruction Scheduling: Reordering of the instruction stream to exploit intra-processor parallelism

For several reasons, e.g. modularization of code generation and portability, code generation may be split into two phases:





### Virtual machine

- idealized architecture,
- simple code generation,
- easily implemented on real hardware.

### Advantages:

- Porting the compiler to a new target architecture is simpler,
- Modularization makes the compiler easier to modify,
- Translation of program constructs is separated from the exploitation of architectural features.



Virtual (or: abstract) machines for some programming languages:

| | | | |
|---|---|---|---|
| Pascal | → | P-machine | |
| Smalltalk | → | Bytecode | |
| Prolog | → | WAM | ("Warren Abstract Machine") |
| SML, Haskell | → | STGM | |
| Java | → | JVM | |



We will consider the following languages and virtual machines:

| | | | | |
|---|---|---|---|---|
| C | → | CMa | // | *imperative* |
| PuF | → | MaMa | // | *functional* |
| Proll | → | WiM | // | *logic based* |
| C± | → | OMa | // | *object oriented* |
| multi-threaded C | → | threaded CMa | // | *concurrent* |



# The Translation of C

# 1 The Architecture of the CMa

- Each virtual machine provides a set of instructions
- Instructions are executed on the virtual hardware
- This virtual hardware can be viewed as a set of data structures, which the instructions access
- ... and which are managed by the run-time system

For the CMa we need:

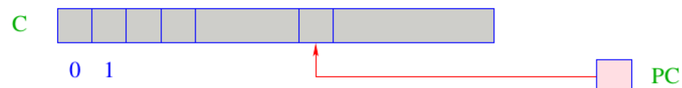


## The Data Store:

- S is the (data) store, onto which new cells are allocated in a LIFO discipline
  $\Longrightarrow$ Stack.
- SP (≘ Stack Pointer) is a register, which contains the address of the topmost allocated cell,
  Simplification: All types of data fit into one cell of S.



## The Code/Instruction Store:




- C is the Code store, which contains the program.
  Each cell of field C can store exactly one virtual instruction.
- PC (≘ Program Counter) is a register, which contains the address of the instruction to be executed next.
- Initially, PC contains the address 0.
  $\Longrightarrow$ C[0] contains the instruction to be executed first.



## Execution of Programs:

- The machine loads the instruction in C[PC] into a Instruction-Register IR and executes it
- PC is incremented by 1 before the execution of the instruction

```
while (true) {
    IR = C[PC]; PC++;
    execute (IR);
}
```

- The execution of the instruction may overwrite the PC (jumps).
- The Main Cycle of the machine will be halted by executing the instruction halt , which returns control to the environment, e.g. the operating system
- More instructions will be introduced by demand

## 2 Simple expressions and assignments

**Problem:** evaluate the expression $(1 + 7) * 3$ !

This means: generate an instruction sequence, which

- determines the value of the expression and
- pushes it on top of the stack...

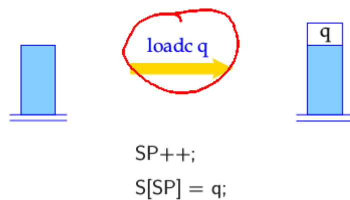
**Idea:**

- first compute the values of the subexpressions,
- save these values on top of the stack,
- then apply the operator.



**The general principle:**

- instructions expect their arguments on top of the stack,
- execution of an instruction consumes its operands,
- results, if any, are stored on top of the stack.

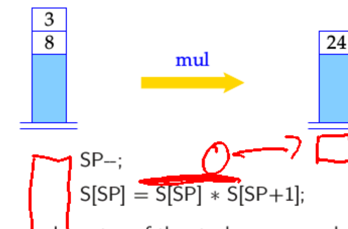



SP++;
S[SP] = q;

Instruction loadc q needs no operand on top of the stack, pushes the constant q onto the stack.

Note: the content of register SP is only implicitly represented, namely through the height of the stack.



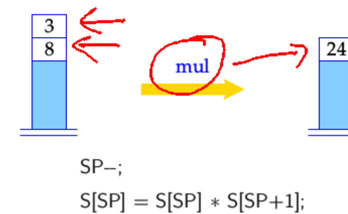



SP−−;
S[SP] = S[SP] ∗ S[SP+1];

mul expects two operands on top of the stack, consumes both, and pushes their product onto the stack.

... the other binary arithmetic and logical instructions, add, sub, div, mod, and, or and xor, work analogously, as do the comparison instructions eq, neq, le, leq, gr and geq.





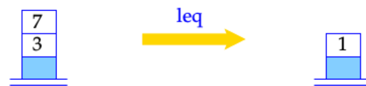

SP−−;
S[SP] = S[SP] ∗ S[SP+1];

mul expects two operands on top of the stack, consumes both, and pushes their product onto the stack.

... the other binary arithmetic and logical instructions, add, sub, div, mod, and, or and xor, work analogously, as do the comparison instructions eq, neq, le, leq, gr and geq.

## Example

The operator leq




Remark: 0 represents *false*, all other integers *true*.

Unary operators neg and not consume one operand and produce one result.

S[SP] = – S[SP];

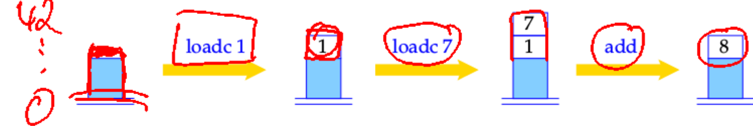


## Example

Code for $1 + 7$:

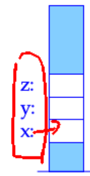
loadc 1    loadc 7    add

Execution of this code sequence:






Variables are associated with memory cells in S:




$\rho$ delivers for each variable $x$ the relative address of $x$.

$\rho$ is called Address Environment.



Variables can be used in two different ways:

## Example

$x = y + 1$

We are interested in the value of $y$, but in the address of $x$.

The syntactic position determines, whether the L-value or the R-value of a variable is required.

L-value of $x$ = address of $x$

R-value of $x$ = content of $x$

| | |
|---|---|
| $\text{code}_R\ e\ \rho$ | produces code to compute the R-value of $e$ in the address environment $\rho$ |
| $\text{code}_L\ e\ \rho$ | analogously for the L-value |

## Note:

Not every expression has an L-value (Ex.: $x + 1$).

We define:

$$\mathrm{code_R}\ (e_1 + e_2)\ \rho \;=\; \mathrm{code_R}\ e_1\ \rho$$
$$\mathrm{code_R}\ e_2\ \rho$$
add

... analogously for the other binary operators

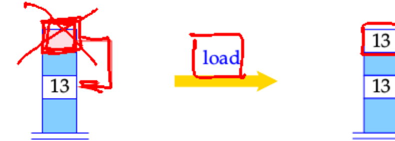
$$\mathrm{code_R}\ (-e)\ \rho \;=\; \mathrm{code_R}\ e\ \rho$$
neg

... analogously for the other unary operators

$$\mathrm{code_R}\ q\ \rho \;=\; \mathrm{loadc}\ q$$
$$\mathrm{code_L}\ x\ \rho \;=\; \mathrm{loadc}\ (\rho\ x)$$
...

$$\mathrm{code_R}\ x\ \rho \;=\; \mathrm{code_L}\ x\ \rho$$
load

The instruction load loads the contents of the cell, whose address is on top of the stack.




S[SP] = S[S[SP]];



$$\mathrm{code_R}\ (x = e)\ \rho \;=\; \mathrm{code_R}\ e\ \rho$$
$$\mathrm{code_L}\ x\ \rho$$
store

store writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

Note: this differs from the code generated by gcc ??

S[S[SP]] = S[SP-1];
SP--;



Example Code for $e \equiv x = y - 1$ with $\rho = \{x \mapsto 4, y \mapsto 7\}$.
$\mathrm{code_R}\ e\ \rho$ produces:

| loadc 7 | loadc 1 | loadc 4 |
|---|---|---|
| load | sub | store |

Improvements:

Introduction of special instructions for frequently used instruction sequences, e.g.,

loada q = loadc q
load

storea q = loadc q
store

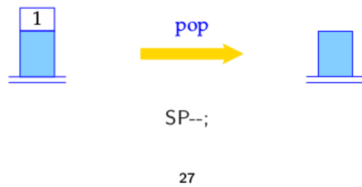
# 3 Statements and Statement Sequences

Is $e$ an expression, then $e;$ is a statement.

Statements do not deliver a value. The contents of the SP before and after the execution of the generated code must therefore be the same.

$$\text{code}\ e;\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{pop}$$

The instruction pop eliminates the top element of the stack.




SP--;



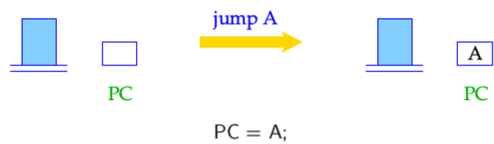
The code for a statement sequence is the concatenation of the code for the statements of the sequence:

$$\text{code}\ (s\ ss)\ \rho \quad = \quad \text{code}\ s\ \rho$$
$$\text{code}\ ss\ \rho$$
$$\text{code}\ \varepsilon\ \rho \quad = \quad // \quad \textit{empty sequence of instructions}$$



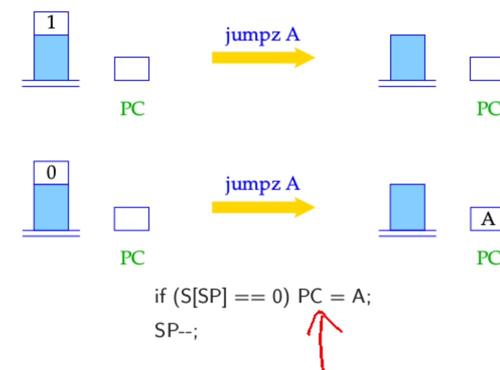
# 4 Conditional and Iterative Statements

We need jumps to deviate from the serial execution of consecutive statements:




PC = A;






if (S[SP] == 0) PC = A;
SP--;

For ease of comprehension, we use symbolic jump targets. They will later be replaced by absolute addresses.

Instead of absolute code addresses, one could generate relative addresses, i.e., relative to the actual PC.

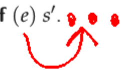
Advantages:

- smaller addresses suffice most of the time;
- the code becomes relocatable, i.e., can be moved around in memory.



## 4.1 One-sided Conditional Statement

Let us first regard $s \equiv \textbf{if } (e) \; s'$.

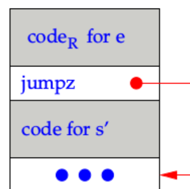


Idea:

- Put code for the evaluation of $e$ and $s'$ consecutively in the code store,
- Insert a conditional jump (jump on zero) in between.



$$
\begin{array}{rcl}
\text{code } s \; \rho & = & \text{code}_R \; e \; \rho \\
 & & \text{jumpz A} \\
 & & \text{code } s' \; \rho \\
\text{A}: & & \ldots
\end{array}
$$

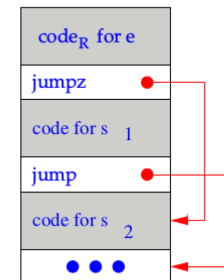





## 4.2 Two-sided Conditional Statement

Let us now regard $s \equiv \textbf{if } (e) \; s_1 \textbf{ else } s_2$. The same strategy yields:

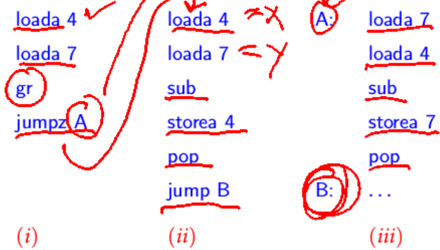
$$
\begin{array}{rcl}
\text{code } s \; \rho & = & \text{code}_R \; e \; \rho \\
 & & \text{jumpz A} \\
 & & \text{code } s_1 \; \rho \\
 & & \text{jump B} \\
\text{A}: & & \text{code } s_2 \; \rho \\
\text{B}: & & \ldots
\end{array}
$$



**Example** Be $\rho = \{x \mapsto 4, y \mapsto 7\}$ and

$$s \equiv \textbf{if } (x > y) \quad (i)$$
$$x = x - y; \quad (ii)$$
$$\textbf{else } y = y - x; \quad (iii)$$

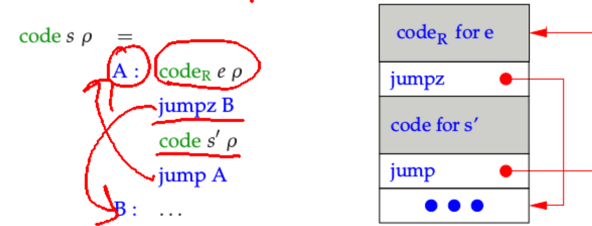
code $s$ $\rho$ produces:






## 4.3 while-Loops

Let us regard the loop $s \equiv \textbf{while } (e)\ s'$. We generate:

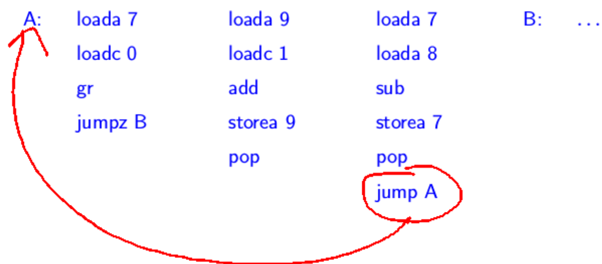





**Example** Be $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ and $s$ the statement:

$$\textbf{while } (a > 0)\ \{c = c + 1;\ a = a - b;\}$$

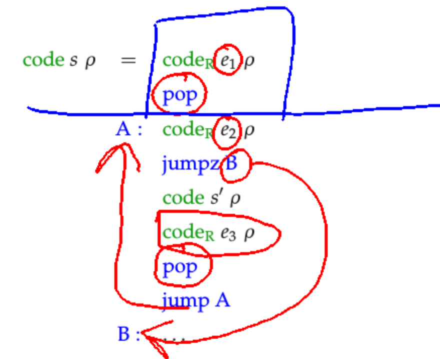
code $s$ $\rho$ produces the sequence:





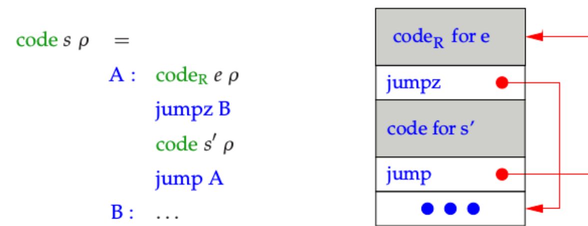

## 4.4 for-Loops

The **for**-loop $s \equiv \textbf{for } (e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;\ \textbf{while } (e_2)\ \{s'\ e_3;\}$ – provided that $s'$ contains no **continue**-statement. We therefore translate:



## 4.3 while-Loops

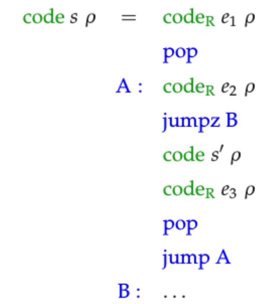
Let us regard the loop $s \equiv$ **while** $(e)\ s'$. We generate:






## 4.4 for-Loops

The **for**-loop $s \equiv$ **for** $(e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;$ **while** $(e_2)\ \{s'\ e_3;\}$ – provided that $s'$ contains no **continue**-statement.
We therefore translate:






VAM