

Script generated by TTT

Title: Seidl: Virtual_machines (17.04.2012)

Date: Tue Apr 17 14:04:37 CEST 2012

Duration: 86:45 min

Pages: 24

Helmut Seidl

Virtual Machines

München

Summer 2012

1

Helmut Seidl

Virtual Machines

München

Summer 2012

1

0 Introduction

Principle of Interpretation:

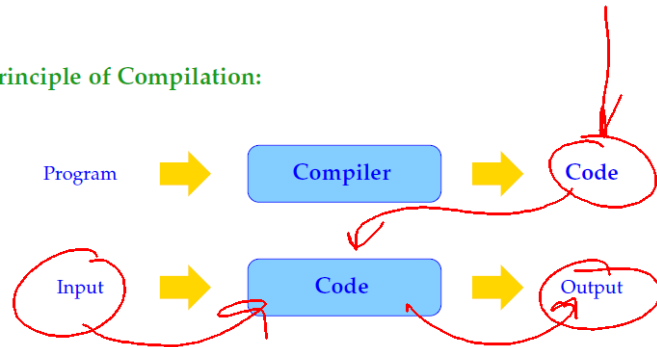


Advantage: No precomputation on the program text \implies no/short startup-time

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

2

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**).

3

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 \implies compilation pays off in long running or often run programs

4

Preprocessing of the source program provides for

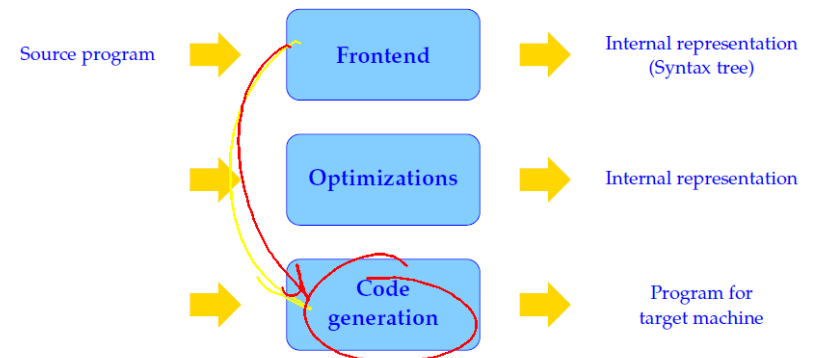
- efficient access to the values of program variables at run time
- global program transformations to increase execution speed.

Disadvantage: Compilation takes time

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

4

Structure of a compiler:



5

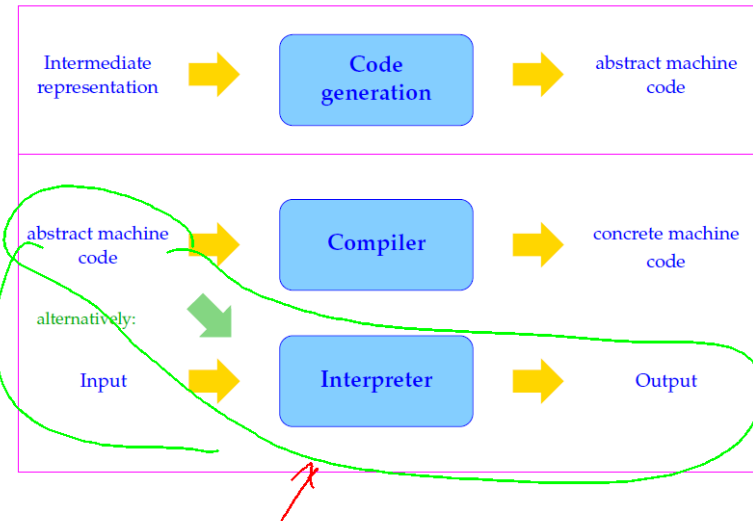
Subtasks in code generation:

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**:

6



7

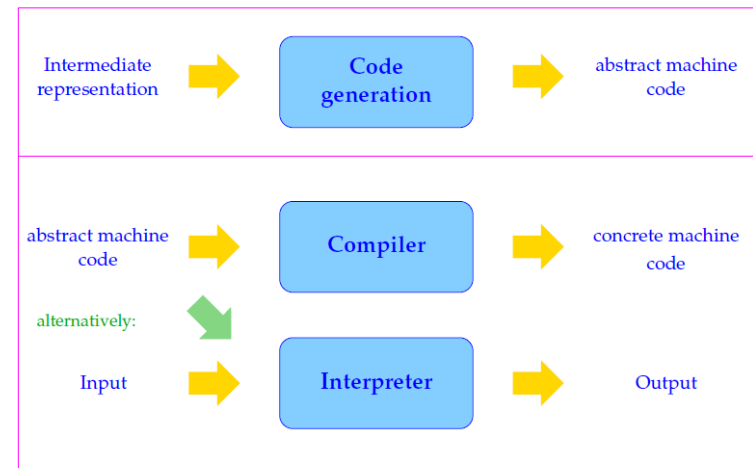
Virtual machine

- 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.

8



7

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.

8

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

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

9

We will consider the following languages and virtual machines:

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

10

The Translation of C

11

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:

12

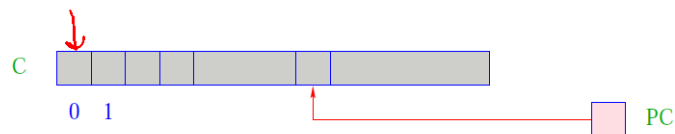
The Data Store:



- **S** is the (data) store, onto which new cells are allocated in a LIFO discipline
⇒ **Stack**.
- **SP** ($\hat{=}$ **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**.

13

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** ($\hat{=}$ **Program Counter**) is a register, which contains the address of the instruction to be executed **next**.
- Initially, **PC** contains the address 0.
⇒ **C[0]** contains the instruction to be executed first.

14

Execution of Programs:

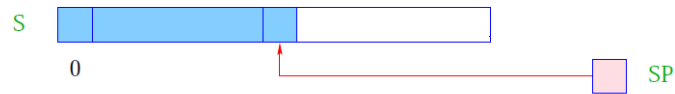
- The machine loads the instruction in **C[PC]** into a **Instruction-Register IR** and executes it
- **PC** is incremented by 1 before the execution of the instruction

```
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**

15

The Data Store:



- S is the (data) store, onto which new cells are allocated in a LIFO discipline
 \implies Stack.
- SP ($\hat{=}$ 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.

13

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..

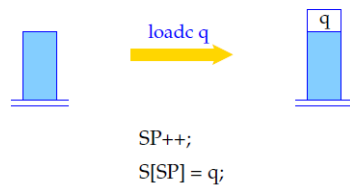
Idea:

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

16

The general principle:

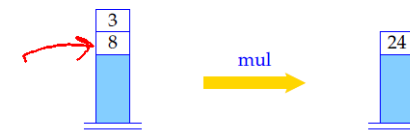
- instructions expect their arguments on top of the stack,
- execution of an instruction consumes its operands,
- results, if any, are stored on top of the stack.



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.

17



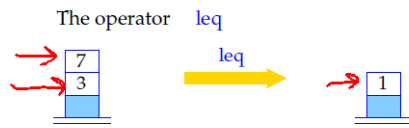
`SP-;`
`S[SP] = S[SP] * S[SP+1];`

`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`.

18

Example:



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

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

