0 Introduction

Principle of Interpretation:

Program + Input \(\rightarrow\) Interpreter \(\rightarrow\) Output

**Advantage:** No precomputation on the program text \(\rightarrow\) no/short startup-time

**Disadvantages:** Program parts are repeatedly analyzed during execution + less efficient access to program variables \(\rightarrow\) 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).

**Structure of a compiler:**

- Source program -> Frontend -> Optimizations -> Code generation
- Internal representation (Syntax tree) -> Internal representation for target machine

**Principle of Interpretation:**

Program + Input -> Interpreter -> Output

**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
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 → STCM
- Java → JVM

We will consider the following languages and virtual machines:

- C → CMa // imperative
- PuF → MaMa // functional
- Prolog → WIM // logic based
- C++ → OMa // object oriented
- multi-threaded C → threaded CMa // concurrent

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.
- \( SP \) (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.
- \( C[0] \) contains the instruction to be executed first.

Execution of Programs:

- The machine loads the instruction in \( C[PC] \) into an 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 \textbf{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.
2 Simple expressions and assignments

Problem: evaluate the expression \((1 + 7) \times 3 \div 1\) !

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.

Instruction **load c** needs no operand on top of the stack, pushes the constant \(c\) onto the stack.

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

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

```java
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