0 Introduction

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 ➞ slower execution speed
**Principle of Compilation:**

Program → Compiler → Code

Input → Code → Output

**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 → compilation pays off in long running or often run programs

**Structure of a compiler:**

Source program → Frontend → Internal representation (Syntax tree)

Optimizations → Internal representation

Code generation → Program for target machine

**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

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
- PaF → 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 CMAs we need:

The Translation of C

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.

2 Simple expressions and assignments

Problem: evaluate the expression $\begin{align*} (1 + 7) \times 3 \end{align*}$

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.

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.

```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.

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.
2 Simple expressions and assignments

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

Instruction \(\text{loadc q}\) 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.

Example: The operator \(\text{leq}\)

Remark: 0 represents false, all other integers true.

Unary operators \(\text{neg}\) and \(\text{not}\) consume one operand and produce one result.

... the other binary arithmetic and logical instructions, \(\text{add}, \text{sub}, \text{div}, \text{mod}, \text{and}, \text{or}\) and \(\text{xor}\), work analogously, as do the comparison instructions \(\text{eq}, \text{neq}, \text{le}, \text{leq}, \text{ge}\) and \(\text{geq}\).
Example: The operator \texttt{leq}

<table>
<thead>
<tr>
<th>7</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>leq</td>
<td>1</td>
</tr>
</tbody>
</table>

Remark: 0 represents \texttt{false}, all other integers \texttt{true}.

Unary operators \texttt{neg} and \texttt{not} consume one operand and produce one result.

<table>
<thead>
<tr>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg</td>
</tr>
</tbody>
</table>

\[ S[SP] = -S[SP] \]

---

Example: Code for \(1 + 7: \)

\[
\text{leade 1} \quad \text{leade 7} \quad \text{add}
\]

Execution of this code sequence:

\[ \text{leade 1} \quad 1 \quad \text{leade 7} \quad 7 \quad 1 \quad \text{add} \quad 8 \]

---

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 \texttt{L-value} or the \texttt{R-value} of a variable is required.

<table>
<thead>
<tr>
<th>\texttt{L-value of } x</th>
<th>=</th>
<th>\texttt{address of } x</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{R-value of } x</td>
<td>=</td>
<td>\texttt{content of } x</td>
</tr>
</tbody>
</table>

\[
\text{co} \varepsilon \rho \quad \text{produces code to compute the R-value of } \varepsilon \text{ in the address environment } \rho
\]

\[
\text{code}_L \varepsilon \rho \quad \text{analogously for the L-value}
\]

Note:

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