Script generated by TTT

Title: Petter: Virtual Machines (29.04.2019)

Date: Mon Apr 29 10:16:17 CEST 2019

Duration: 94:42 min

Pages: 12

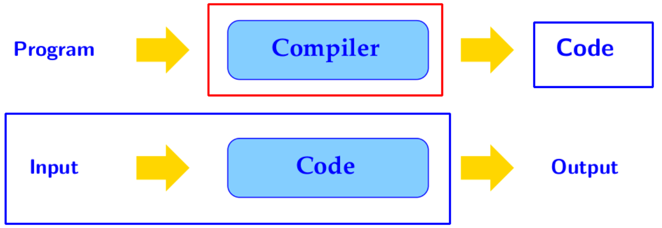
Helmut Seidl, Michael Petter

Virtual Machines

München

Summer 2019

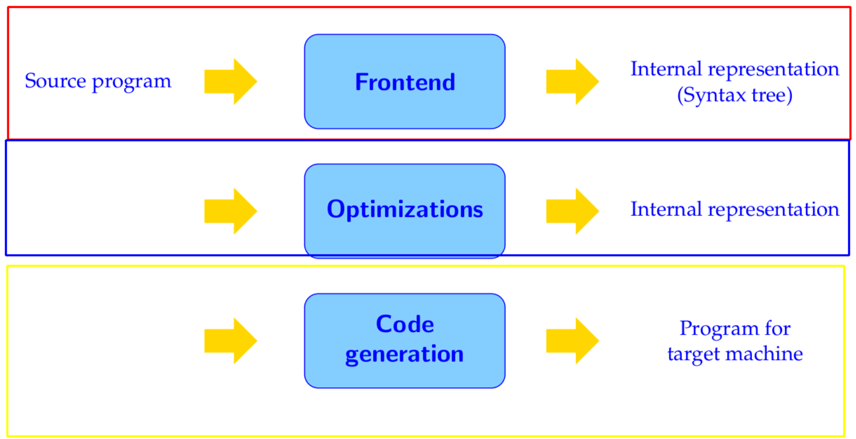
Principle of Compilation



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:

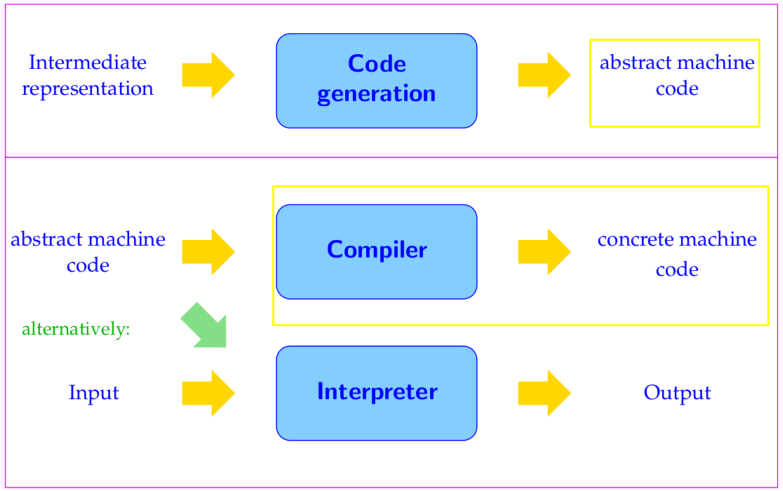


Subtasks in code generation:

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.

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

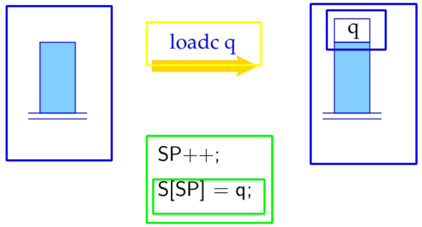
- determines the value of the expression and
- pushes it on top of the stack...

Idea

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

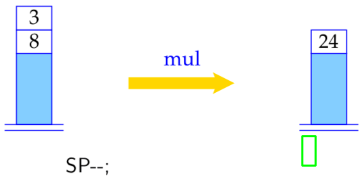
The general principle

- 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.



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

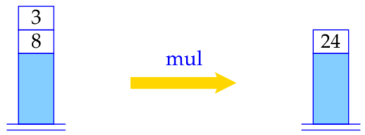
The content of register SP is only implicitly represented, namely, through the height of the stack.



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.

We define:

$$\text{code}_R\ (e_1 + e_2)\ \rho \quad = \quad \text{code}_R\ e_1\ \rho$$
$$\text{code}_R\ e_2\ \rho$$
$$\text{add}$$

... analogously for the other binary operators

$$\text{code}_R\ (-e)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{neg}$$

... analogously for the other unary operators

$$\text{code}_R\ q\ \rho \quad = \quad \text{loadc q}$$
$$\text{code}_L\ x\ \rho \quad = \quad \text{loadc}\ (\rho\ x)$$

...