Generating Code: Overview

We inductively generate instructions from the AST:
- there is a rule stating how to generate code for each non-terminal of the grammar
- the code is merely another attribute in the syntax tree
- code generation makes use of the already computed attributes

In order to specify the code generation, we require
- a semantics of the language we are compiling (here: C standard)
- the semantic of the machine instructions
Generating Code: Overview

We inductively generate instructions from the AST:
- there is a rule stating how to generate code for each non-terminal of the grammar
- the code is merely another attribute in the syntax tree
- code generation makes use of the already computed attributes

In order to specify the code generation, we require
- a semantics of the language we are compiling (here: C standard)
- the semantic of the machine instructions

we commence by specifying machine instruction semantics

The Register C-Machine (RCMa)

We generate Code for the Register C-Machine. The Register C-Machine is a virtual machine (VM).
- there exists no processor that can execute its instructions
- ... but we can build an interpreter for it
- we provide a visualization environment for the R-CMa
- the R-CMa has no double, float, char, short or long types
- the R-CMa has no instructions to communicate with the operating system
- the R-CMa has an unlimited supply of registers

The R-CMa is more realistic than it may seem:
- the mentioned restrictions can easily be lifted
- the Java virtual machine (JVM) is similar to the R-CMa but has no registers
- an interpreter of R-CMA can run on any platform

Virtual Machines

A virtual machine has the following ingredients:
- any virtual machine provides a set of instructions
- instructions are executed on virtual hardware
- the virtual hardware is a collection of data structures that is accessed and modified by the VM instructions
- ... and also by other components of the run-time system, namely functions that go beyond the instruction semantics
- the interpreter is part of the run-time system

Components of a Virtual Machine

Consider Java as an example:

A virtual machine such as the JVM has the following structure:
- S: the data store – a memory region in which cells can be stored in LIFO order ~ stack.
- SP: (= stack pointer) pointer to the last used cell in S
- beyond S, the memory containing the heap follows
Components of a Virtual Machine
Consider Java as an example:

A virtual machine such as the JVM has the following structure:
- **S**: the data store – a memory region in which cells can be stored in LIFO order – stack.
- **SP**: (≡ stack pointer) pointer to the last used cell in S
- beyond S, the memory containing the heap follows
- **C** is the memory storing **code**
  - each cell of C holds exactly one virtual instruction
  - C can only be read
- **PC** (≡ program counter) address of the instruction that is to be executed next
- **PC** contains 0 initially

Executing a Program

- the machine loads an instruction form C[PC] into an instruction register IR in order to execute it
- before evaluating the instruction, the **PC** is incremented by one

```java
while (true) {
    IR = C[PC]; PC++;
    execute (IR);
}
```
- node: the **PC** must be incremented before the execution, since an instruction may modify the **PC**
- the loop is exited by evaluating a **halt** instruction that returns directly to the operating system

Simple Expressions and Assignments

**Chapter 2: Evaluation of Expressions**

**Task**: evaluate the expression \((1 + 7) \times 3\) that is, generate an instruction sequence that
- computes the value of the expression and
- stores it on top of the stack
Simple Expressions and Assignments

Task: evaluate the expression

that is, generate an instruction sequence that
- computes the value of the expression and
- stores it on top of the stack

Idea:
- first compute the value of the sub-expressions
- store the intermediate result on top of the stack
- apply the operator

General Principle

Evaluating an operation $\text{op}(a_1, \ldots, a_n)$
- the arguments $a_1, \ldots, a_n$ must be on top of the stack
- the execution of the operation $\text{op}$ consumes its arguments
- any resulting values are stored on top of the stack

```
iconst q
```

```
SP++; S[SP] = q;
```

$q \in \mathbb{Z}$

the instruction $\text{iconst} \ q$ puts the int-constant $q$ onto the stack

Binary Operators

Operators with two arguments run as follows:

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

$\text{imul}$
Composition of Instructions

Example: generate code for $1 + 7$:

- $iconst\ 1$
- $iconst\ 7$
- $iadd$

Execution of this instruction sequence:

Expressions with Variables

Variables occupy a memory cell in $S$:

- Associating addresses with variables can be done while creating the symbol table. The address is stored in any case at the node of the declaration of a variable.

Expressions with Variables

Variables occupy a memory cell in $S$:

- Associating addresses with variables can be done while creating the symbol table. The address is stored in any case at the node of the declaration of a variable.
- For each use of a variable, the address has to be looked up by inspecting its declaration node.
Expressions with Variables

Variables occupy a memory cell in $S$:

- Associating addresses with variables can be done while creating the symbol table. The address is stored in any case at the node of the declaration of a variable.
- For each use of a variable, the address has to be looked up by inspecting its declaration node.
- In the sequel, we use a mathematical map $\rho$, that contains mappings form a variable $x$ to the (relative) address of $x$; the map $\rho$ is called address environment (or simply environment).

Reading from a Variable

The instruction $i\text{load } k$ loads the value at address $k$, where $k$ is relative to the top of the stack.

$S[SP+1] = S[SP-k]; SP = SP+1;$

Example: Compute $x + 2$ where $\rho = \{x \mapsto 1\}$:

Code Synthesis

Chapter 3:
Generating Code for the Register C-Machine
Motivation for the Register C-Machine

A modern RISC processor features a fixed number of universal registers.

- arithmetic operations can only use these registers as arguments
- access to memory are done via instructions to load and store to and from registers
- unlike the stack, registers have to be explicitly saved before a function is called

A translation for a RISC processor must therefore:

- store variables and function arguments in registers
- save the content of registers onto the stack before calling a function
- express any arbitrary computation using finitely many registers

~ only consider the first two problems (and deal with the other two later)
Principle of the Register C-Machine

The R-CMa is composed of a stack, heap and a code segment, just like the JVM; it additionally has register sets:
- **local** registers are $R_1, R_2, \ldots R_i, \ldots$
- **global** register are $R_0, R_{-1}, \ldots R_j, \ldots$

The Register Sets of the R-CMa

The two register sets have the following purpose:
- the **local** registers $R_i$
  - save temporary results
  - store the contents of local variables of a function
  - can efficiently be stored and restored from the stack
- the **global** registers $R_i$
  - save the parameters of a function
  - store the result of a function

Note:
for now, we only use registers to store temporary computations
The Register Sets of the R-CMa

The two register sets have the following purpose:

- the *local* registers $R_i$
  - save temporary results
  - store the contents of local variables of a function
  - can efficiently be stored and restored from the stack
- the *global* registers $R_i$
  - save the parameters of a function
  - store the result of a function

Note:
for now, we only use registers to store temporary computations

Idea for the translation: use a register counter $i$:
- registers $R_j$ with $j < i$ are *in use*
- registers $R_j$ with $j \geq i$ are *available*

Translation of Simple Expressions

Using variables stored in registers; loading constants:

<table>
<thead>
<tr>
<th>instruction</th>
<th>semantics</th>
<th>intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $R_c$, $c$</td>
<td>$R_c = c$</td>
<td>load constant</td>
</tr>
<tr>
<td>move $R_i$, $R_j$</td>
<td>$R_i = R_j$</td>
<td>copy $R_j$ to $R_i$</td>
</tr>
</tbody>
</table>

We define the following translation schema (with $\rho x = a$):

\[
\begin{align*}
\text{code}_R c \rho &= \text{load } R_c \ c \\
\text{code}_R x \rho &= \text{move } R_i, R_j \\
\text{code}_R x = e \rho &= \text{code}^i_R e \rho \\
\text{code}_R x = e \rho &= \text{move } R_i, R_j \\
\text{code}_R x = e \rho &= \text{load } R_c \ c \\
\text{code}_R x = e \rho &= \text{move } R_i, R_j \\
\text{code}_R x = e \rho &= \text{move } R_i, R_j
\end{align*}
\]

Translation of Simple Expressions

Using variables stored in registers; loading constants:

<table>
<thead>
<tr>
<th>instruction</th>
<th>semantics</th>
<th>intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $R_c$, $c$</td>
<td>$R_c = c$</td>
<td>load constant</td>
</tr>
<tr>
<td>move $R_i$, $R_j$</td>
<td>$R_i = R_j$</td>
<td>copy $R_j$ to $R_i$</td>
</tr>
</tbody>
</table>

We define the following translation schema (with $\rho x = a$):

\[
\begin{align*}
\text{code}_R^i c \rho &= \text{load } R_c \ c \\
\text{code}_R^i x \rho &= \text{move } R_i, R_a \\
\text{code}_R^i x = e \rho &= \text{code}^i_R e \rho \\
\text{code}_R^i x = e \rho &= \text{move } R_i, R_a \\
\text{code}_R^i x = e \rho &= \text{move } R_i, R_a
\end{align*}
\]

Note: all instructions use the Intel convention (in contrast to the AT&T convention): \( op \ dst \ src_1 \ldots \ src_n \).
**Translation of Expressions**

Let \( \text{op} = \{\text{add}, \text{sub}, \text{div}, \text{mul}, \text{mod}, \text{le}, \text{gr}, \text{eq}, \text{leq}, \text{geq}, \text{and}, \text{or}\} \). The R-CMa provides an instruction for each operator \( \text{op} \).

\[
\text{op} \ R_i \ R_j \ R_k
\]

where \( R_i \) is the target register, \( R_j \) the first and \( R_k \) the second argument.

Correspondingly, we generate code as follows:

\[
\text{code}^i_{R} e_1 \ \text{op} \ e_2 \rho = \text{code}^{i+1}_{R} e_1 \rho
\]

\[
\text{op} \ R_i \ R_j \ R_{k+1}
\]

**Example:** Translate \( 3 \times 4 \) with \( i = 4 \):

\[
\text{code}^i_{R} 3 \times 4 \rho = \text{load} \ R_5 \ 3
\]

\[
\text{mul} \ R_i \ R_k \ R_5
\]
Translation of Expressions

Let \( \text{op} = \{ \text{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or} \} \). The R-CMA provides an instruction for each operator \( \text{op} \).

\[
\text{add}(R_i, \text{lo} + 1, R_j, R_k)
\]

where \( R_i \) is the target register, \( R_j \) the first and \( R_k \) the second argument.

Correspondingly, we generate code as follows:

\[
\text{code}^{\text{R}}_k(e_1 \; \text{op} \; e_2 \; \rho) = \text{code}^{\text{R}}_k(e_1 \; \rho) \oplus \text{code}^{\text{R}}_{k+1}(e_2 \; \rho) \oplus \text{op}(R_i, R_j, R_k) + \text{lo}(R_i, R_j, R_k)
\]

we obtain

\[
\text{code}^{\text{R}}_k(3 \times 4 + 3 \times 4 \; \rho) = \text{load}(R_1, 3) \; \text{load}(R_2, 4) \; \text{mul}(R_3, R_4, R_5) \; \text{add}(R_4, R_5, R_6)
\]

Managing Temporary Registers

Observe that temporary registers are re-used: translate \( 3 \times 4 + 3 \times 4 \) with \( \rho = 4 \):

\[
\text{code}^{\text{R}}_k(3 \times 4 \; \rho) = \text{code}^{\text{R}}_k(3 \times 4 \; \rho) \oplus \text{add}(R_4, R_5, R_6)
\]

where

\[
\text{code}^{\text{R}}_k(3 \times 4 \; \rho) = \text{load}(R_3, 3) \; \text{load}(R_4, 4) \; \text{mul}(R_3, R_4, R_5) \; \text{add}(R_4, R_5, R_6)
\]

Semantics of Operators

The operators have the following semantics:

- **add** \( R_i, R_j, R_k \) \( R_i = R_j + R_k \)
- **sub** \( R_i, R_j, R_k \) \( R_i = R_j - R_k \)
- **div** \( R_i, R_j, R_k \) \( R_i = R_j / R_k \)
- **mul** \( R_i, R_j, R_k \) \( R_i = R_j \times R_k \)
- **mod** \( R_i, R_j, R_k \) \( R_i = \text{sgn}(R_i)k \) wobei \( |R_j| = \lfloor n|R_i| + k \rfloor \wedge 0 \leq k < |R_i| \)
- **le** \( R_i, R_j, R_k \) \( R_i = \text{if } R_i < R_j \text{ then } 1 \text{ else } 0 \)
- **gr** \( R_i, R_j, R_k \) \( R_i = \text{if } R_i > R_j \text{ then } 1 \text{ else } 0 \)
- **eq** \( R_i, R_j, R_k \) \( R_i = \text{if } R_i = R_j \text{ then } 1 \text{ else } 0 \)
- **leq** \( R_i, R_j, R_k \) \( R_i = \text{if } R_i \leq R_j \text{ then } 1 \text{ else } 0 \)
- **geq** \( R_i, R_j, R_k \) \( R_i = \text{if } R_i \geq R_j \text{ then } 1 \text{ else } 0 \)
- **and** \( R_i, R_j, R_k \) \( R_i = R_j \& R_k \) // bit-wise and
- **or** \( R_i, R_j, R_k \) \( R_i = R_j \| R_k \) // bit-wise or

Note: all registers and memory cells contain operands in \( \mathbb{Z} \)
Translation of Unary Operators

Unary operators $\text{op} = \{\text{neg, not}\}$ take only two registers:

$$\text{code}^{\text{op}}_{\text{R}} e \rho_{\text{op}} = \text{code}^{\text{op}}_{\text{R}} e \rho_{\text{op}} \text{ op} \ R_{\text{R}}, \ R_{\text{R}}$$

Note: We use the same register.

Example: Translate $-4$ into $R_5$:

$$\text{code}^{5}_{R} -4 \ \rho = \text{load} \ R_5 \ 4 \ \neg \ R_2 \ R_5$$

Translation of Unary Operators

Unary operators $\text{op} = \{\text{neg, not}\}$ take only two registers:

$$\text{code}^{\text{op}}_{\text{R}} e \rho_{\text{op}} = \text{code}^{\text{op}}_{\text{R}} e \rho_{\text{op}} \text{ op} \ R_{\text{R}}, \ R_{\text{R}}$$

Note: We use the same register.

Applying Translation Schema for Expressions

Suppose the following function is given:

```c
void \( f(\text{void}) \) {
    \text{int } x, y, z;
    x = y + z + 3;
}
```

- Let $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ be the address environment.
- Let $R_4$ be the first free register, that is, $i = 4$.

$$\text{code}^{4}_{R} x = y + z + 3 \ \rho = \text{code}^{4}_{R} y + z + 3 \ \rho \text{ move} \ R_1 \ R_4$$

$\rho(x) = 1$
Applying Translation Schema for Expressions
Suppose the following function is given:

```c
void f(void) {
    int x, y, z;
    x = y+z*3;
}
```

- Let $\rho = \{ x \mapsto 1, y \mapsto 2, z \mapsto 3 \}$ be the address environment.
- Let $R_4$ be the first free register, that is, $i = 4$.

\[
\begin{align*}
\text{code}^4 & \quad x = y + z \times 3 &\quad \text{code}^4 & \quad x = y + z \times 3 \\
\text{move} & \quad R_1, R_4 &\quad \text{move} & \quad R_1, R_4 \\
\text{code}^4 & \quad y + z \times 3 &\quad \text{code}^4 & \quad y + z \times 3 \\
\text{move} & \quad R_4, R_2 &\quad \text{move} & \quad R_4, R_2 \\
\text{code}^5 & \quad z \times 3 &\quad \text{code}^5 & \quad z \times 3 \\
\text{add} & \quad R_4, R_5 &\quad \text{add} & \quad R_4, R_5 \\
\text{code}^5 & \quad z + 3 &\quad \text{code}^5 & \quad z + 3 \\
\text{load} & \quad R_6 &\quad \text{load} & \quad R_6 \\
\text{mul} & \quad R_5, R_6 &\quad \text{mul} & \quad R_5, R_6 \\
\text{code}^5 & \quad 3 &\quad \text{code}^5 & \quad 3 \\
\text{load} & \quad R_0 &\quad \text{load} & \quad R_0 \\
\end{align*}
\]

Applying Translation Schema for Expressions
Suppose the following function is given:

```c
void f(void) {
    int x, y, z;
    x = y + z * 3;
}
```

- Let $\rho = \{ x \mapsto 1, y \mapsto 2, z \mapsto 3 \}$ be the address environment.
- Let $R_4$ be the first free register, that is, $i = 4$.

\[
\begin{align*}
\text{code}^4 & \quad x = y + z \times 3 &\quad \text{code}^4 & \quad x = y + z \times 3 \\
\text{move} & \quad R_1, R_4 &\quad \text{move} & \quad R_1, R_4 \\
\text{code}^4 & \quad y + z \times 3 &\quad \text{code}^4 & \quad y + z \times 3 \\
\text{move} & \quad R_4, R_2 &\quad \text{move} & \quad R_4, R_2 \\
\text{code}^5 & \quad z \times 3 &\quad \text{code}^5 & \quad z \times 3 \\
\text{add} & \quad R_4, R_5 &\quad \text{add} & \quad R_4, R_5 \\
\text{code}^5 & \quad z + 3 &\quad \text{code}^5 & \quad z + 3 \\
\text{load} & \quad R_6 &\quad \text{load} & \quad R_6 \\
\text{mul} & \quad R_5, R_6 &\quad \text{mul} & \quad R_5, R_6 \\
\text{code}^5 & \quad 3 &\quad \text{code}^5 & \quad 3 \\
\text{load} & \quad R_0 &\quad \text{load} & \quad R_0 \\
\end{align*}
\]

~\text{the assignment } x = y + z \times 3 \text{ is translated as}

\[
\text{move} \quad R_4, R_2; \text{move} \quad R_5, R_3; \text{load} \quad R_6; \text{mul} \quad R_5, R_6; \text{add} \quad R_4, R_5; \text{move} \quad R_1, R_4
\]

Chapter 4:
Statements and Control Structures
About Statements and Expressions

General idea for translation:
\[ \text{code}^t_s \rho : \text{generate code for statement } s \]
\[ \text{code}^t_e \rho : \text{generate code for expression } e \text{ into } R_i \]
Throughout: \( i, i + 1, \ldots \) are free (unused) registers

For an expression \( x = e \) with \( \rho x = a \) we defined:
\[ \text{code}^t_R x = e \rho = \text{code}^t_R e \rho \]
move \( R_i \)

However, \( x = e \) is also a statement:

About Statements and Expressions

General idea for translation:
\[ \text{code}^t_s \rho : \text{generate code for statement } s \]
\[ \text{code}^t_e \rho : \text{generate code for expression } e \text{ into } R_i \]
Throughout: \( i, i + 1, \ldots \) are free (unused) registers

For an expression \( x = e \) with \( \rho x = a \) we defined:
\[ \text{code}^t_R x = e \rho = \text{code}^t_R e \rho \]
move \( R_i \), \( R_i \)

However, \( x = e \) is also a statement:

- Define:
\[ \text{code}^t e_1 = e_2 \rho = \text{code}^t_R e_1 = e_2 \rho \]

The temporary register \( R_i \) is ignored here. More general:
\[ \text{code}^t e \rho = \text{code}^t_R e \rho \]

Jumps

In order to diverge from the linear sequence of execution, we need jumps:

![Jumps Diagram]

The temporary register \( R_i \) is ignored here. More general:
\[ \text{code}^t e \rho = \text{code}^t_R e \rho \]

- Observation: the assignment to \( e_1 \) is a side effect of the evaluating the expression \( e_1 = e_2 \).
Conditional Jumps

A conditional jump branches depending on the value in \( R_i \):

\[
\text{ unconditional jumps through } \text{ near jumps (if applicable)}
\]

Management of Control Flow

In order to translate statements with control flow, we need to emit jump instructions:

- during the translation of an \( \text{ if } (c) \) construct, it is not yet clear where to jump to in case that \( c \) is false
- instruction sequences may be arranged in a different order
  - minimize the number of \text{ unconditional jumps}
  - minimize in a way so that fewer jumps are executed inside loops
  - replace \text{ far jumps} through \text{ near jumps} (if applicable)

- organize instruction sequence into blocks without jumps
Basic Blocks and the Register C-Machine

The R-CMa features only a single conditional jump, namely `jumpz`.

Outgoing edges must have the following form:

1. a single edge (unconditional jump), translated with `jump`
2. two edges, one with `c = 0` as condition and one without condition, translated with `jumpz` and `jump`, respectively
3. a set of edges and one `default` edge, used for `switch` statement, translated with `jumpl` and `jump` (to be discussed later)

---

Basic Blocks and the Register C-Machine

The R-CMa features only a single conditional jump, namely `jumpz`.

Outgoing edges must have the following form:

- a single edge (unconditional jump), translated with `jump`

---

Formalizing the Translation Involving Control Flow

For simplicity of defining translations of instructions involving control flow, we use symbolic jump targets.

- This translation can be used in practice, but a second run through the emitted instructions is necessary to `resolve` the symbolic addresses to actual addresses.
Formalizing the Translation Involving Control Flow
For simplicity of defining translations of instructions involving control flow, we use **symbolic jump targets.**

- This translation can be used in practice, but a second run through the emitted instructions is necessary to **resolve** the symbolic addresses to actual addresses.

Alternatively, we can emit **relative jumps without a second pass:**
- relative jumps have targets that are offsets to the current **PC**
- sometime relative jumps only possible for small offsets (→ near jumps)
- if all jumps are relative: the code becomes **position independent** (PIC), that is, it can be moved to a different address
- the generated code can be **loaded without relocating absolute jumps**

> generating a graph of basic blocks is useful for **program optimization** where the statements inside basic blocks are simplified

Simple Conditional
We first consider \( s \equiv \text{if} (c) \text{ss} \).

- and present a translation without basic blocks.

**Idea:**
- emit the code of \( c \) and \( ss \) in sequence
- insert a jump instruction in-between, so that correct control flow is ensured

\[
\text{code}^t_s \rho = \begin{array}{l}
\text{code}^R_c \rho \\
\text{jumpz} \, R, A \\
\text{code}^t_{ss} \rho \\
\end{array}
\]

\[A: \quad \ldots\]

General Conditional
Translation of \( \text{if} (c) \text{tt else ee} \).

\[
\text{code}^t \text{if}(c) \text{tt else ee} \rho = \begin{array}{l}
\text{code}^R_e \rho \\
\text{jumpz} \\
\text{code}^t \text{tt} \rho \\
\text{jumpz} \, R, A \\
\text{code}^t_{ee} \rho \\
\end{array}
\]

\[A: \quad \ldots\]

\[B: \quad \ldots\]
Example for if-statement

Let $\rho = \{ x \mapsto 4, y \mapsto 7 \}$ and let $s$ be the statement

```c
if (x>y) {
    x = x - y;  /* (i) */
} else {
    y = y - x;  /* (ii) */
}
```

Then `code' $s \rho$ yields:

```c
\text{code} x
\text{code} y
q \rightarrow R_i R_i R_{i+1}
jump R_i A
\text{code} x = x - y
jump B
A: \text{code} y = y - x
B:
```