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)
a semantics of the machine instructions

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)
a semantics of the machine instructions
\[\text{we commence by specifying machine instruction semantics} \]
The Register C-Machine (R-CMa)

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 `Dalvik VM` or the `LLVM` are similar to the R-CMa
- 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 Dalvik VM has the following structure:
- S: the data store — a memory region in which cells can be stored in LIFO order \(\sim\) stack.
- SP: \(\equiv\) stack pointer) pointer to the last used cell in S beyond S follows the memory containing the heap

Executing a Program

the machine loads an instruction from \(C[PC]\) into the instruction register IR in order to execute it.
before evaluating the instruction, the PC is incremented by one

\[
\text{while (true)} \{ \\
    \text{IR} = C[\text{PC}]; \text{PC}++; \\
    \text{execute (IR);} \\
\}
\]

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

Chapter 2:
Generating Code for the Register C-Machine
Simple Expressions and Assignments in R-CMa

Task: evaluate the expression \((1 + 7) + 3\)
that is, generate an instruction sequence that
computes the value of the expression and
keeps its value accessible in a reproducible way.

Idea:
first compute the value of the sub-expressions
store the intermediate result in a temporary register
apply the operator
loop

Principles of the R-CMa

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** registers 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 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

Translation of Simple Expressions

Using variables stored in registers; loading constants:

\[
\begin{align*}
\text{load } R_i \leftarrow e & \quad R_i = e & \quad \text{load constant} \\
\text{move } R_i & \leftarrow R_j & \quad \text{copy } R_j \text{ to } R_i
\end{align*}
\]

\[\text{Instr} \rightarrow \text{dst} \rightarrow \text{src_1} \rightarrow \ldots \rightarrow \text{src_n}\]
Translation of Simple Expressions

Using variables stored in registers; loading constants:

\[
\begin{align*}
\text{load} & : R_i, c & R_i = c & \text{load constant} \\
\text{move} & : R_i, R_j & R_i = R_j & \text{copy } R_j \text{ to } R_i
\end{align*}
\]

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

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

Translation of Expressions

Let \( \text{op} = \{\text{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or}\} \).
The \text{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:

\[
\begin{align*}
\text{code}_R^i \ e_1 \ \text{op} \ e_2 \ \rho & = \ \text{code}_R^i (e_1 \ \rho) \\
\text{code}_R^{i+1} \ e_2 \ \rho & \ \text{op} \ R_i, R_i, R_{i+1}
\end{align*}
\]

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

\[
\begin{align*}
\text{code}_R^i \ 5 \times 4 & = \ \text{code}_R^i \ 5 \ \rho \\
\text{mul} & : R_i, R_j
\end{align*}
\]
Managing Temporary Registers

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

\[
\text{code}^{\mu}_R \ 3 \times 4 + 3 \times 4 \ \rho = \ \text{code}^{\mu}_R \ 3 \times 4 \ \rho \\
\text{add} \ R_4 \ R_4 \ R_4
\]

where

\[
\text{code}^{\mu}_R \ 3 \times 4 \ \rho = \ \text{load} \ R_3 \\
\text{load} \ R_{i+1} \ 4 \\
\text{mul} \ R_0 \ R_i \ R_{i+1}
\]

we obtain

\[
\text{code}^{\mu}_R \ 3 \times 4 + 3 \times 4 \ \rho = \ \text{load} \ R_3 \\
\text{load} \ R_{i+1} \ 4 \\
\text{mul} \ R_0 \ R_i \ R_{i+1}
\]

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 \(R_i, R_j, R_k\) \(R_i = \text{signum}(R_k) \cdot k\) with \(|R_i| = n \cdot |R_k| + k \land n > 0, 0 < k < |R_k|\)
- le \(R_i, R_j, R_k\) \(R_i = \text{if } R_j < R_k \text{ then } 1 \text{ else } 0\)
- gr \(R_i, R_j, R_k\) \(R_i = \text{if } R_j > R_k \text{ then } 1 \text{ else } 0\)
- eq \(R_i, R_j, R_k\) \(R_i = \text{if } R_j = R_k \text{ then } 1 \text{ else } 0\)
- leq \(R_i, R_j, R_k\) \(R_i = \text{if } R_j \leq R_k \text{ then } 1 \text{ else } 0\)
- geq \(R_i, R_j, R_k\) \(R_i = \text{if } R_j \geq R_k \text{ then } 1 \text{ else } 0\)
- and \(R_i, R_j, R_k\) \(R_i = R_j \& R_k\) // bit-wise and
- or \(R_i, R_j, R_k\) \(R_i = R_j \mid R_k\) // bit-wise or

Translation of Unary Operators

Unary operators \(\text{op} = \begin{cases} \text{neg} & \text{not} \end{cases}\) take only two registers:

\[
\text{code}^{\mu}_R \ \text{op} \ \rho = \ \text{code}^{\mu}_R \ e \ \rho \\
\text{op} \ R_i \ R_i
\]
Translation of Unary Operators

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

\[
\text{code}_R^\text{h} \text{ op } e \rho = \text{code}_R^\text{r} \text{ e } \rho \\
\text{op } R_i, R_i
\]

**Note:** We use the same register.

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

\[
\text{code}_R^\text{h} -4 \rho = \text{code}_R^\text{h} 4 \rho \\
\text{neg } R_5, R_5
\]

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 $R_1$ be the first free register, that is, $i = 4$.

\[
\text{Code } x = y + z + 3 \rho = \text{code}_R^\text{h} y + z + 3 \rho \\
\text{move } R_1, R_4
\]

Translation of Unary Operators

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

\[
\text{code}_R^\text{r} \text{ op } e \rho = \text{code}_R^\text{r} \text{ e } \rho \\
\text{op } R_i, R_i
\]

**Note:** We use the same register.

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

\[
\text{code}_R^\text{r} -4 \rho = \text{load } R_5, 4 \\
\text{neg } R_5, R_5
\]

The operators have the following semantics:

\[
\text{not } R_i, R_j \rightarrow R_i \left\{ \begin{array}{l}
\text{if } R_j = 0 \text{ then } 1 \text{ else } 0
\end{array} \right.
\]

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 $R_1$ be the first free register, that is, $i = 4$.

\[
\text{Code } x = y + z + 3 \rho = \text{code}_R^\text{h} y + z + 3 \rho \\
\text{move } R_1, R_4
\]

\[
\text{Code } z + 3 \rho = \text{code}_R^\text{h} z + 3 \rho \\
\text{add } R_4, R_3, R_5
\]
Applying Translation Schema for Expressions
Suppose the following function

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

is given:

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

\[
\begin{align*}
\text{code}_R^4 \ x &= y + z + 3 \ \rho \\
\text{code}_R^4 \ y &= y + z + 3 \ \rho \\
\text{move} \ R_1 \ R_4 \\
\end{align*}
\]

\[
\begin{align*}
\text{code}_R^5 \ y &= y + z + 3 \ \rho \\
\text{code}_R^5 \ z &= z + 3 \ \rho \\
\text{add} \ R_4 \ R_4 \ R_5 \\
\end{align*}
\]

\[
\begin{align*}
\text{code}_R^6 \ R_3 &= \text{move} \\
\text{code}_R^6 \ R_5 &= \text{mul} \\
\text{code}_R^6 \ R_6 &= \text{load} \ R_6 \ 3
\end{align*}
\]


Chapter 3:
Statements and Control Structures
About Statements and Expressions

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

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

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

Define:
\[
\text{code}^i_e \ e_1 = e_2; \ \rho \quad = \quad \text{code}^i_e \ e_1 = e_2 \ \rho
\]

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

Translation of Statement Sequences

The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:
\[
\text{code}^i [s \ s s] \ \rho \quad = \quad \text{code}^i_s \ \rho
\]
\[
\text{code}^i \emptyset \ \rho \quad = \quad \emptyset \quad \text{empty sequence of instructions}
\]

Note here: \(s\) is a statement, \(ss\) is a sequence of statements
In order to diverge from the linear sequence of execution, we need jumps:

\[ \text{PC} = A, \]

**Simple Conditional**

We first consider

\[ \text{if (c)} \{
\text{ss};
\}\]

...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

\[
\text{code}^f \ s \ p = \begin{cases} 
\text{code}^f \ c \ p \\
\text{jumpz} \ 
\text{Ri} \ A \\
\text{code}^f \ ss \ p \\
A \\
\ldots
\end{cases}
\]

**Conditional Jumps**

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

\[ \text{if (} R_i = 0 \text{)} \ 	ext{PC} = A; \]