Overloading and Coercion

Some operators such as + are **overloaded**:
- + has **several possible** types
  - for example: `int + (int, int), float + (float, float)`
  - but also `float* + (float*, int), int* + (int, int*)`
- depending on the type, the operator + has a different implementation
- determining which implementation should be used is based on the *arguments* only

Coercion: allow the application of + to int and float.
- instead of defining + for all possible combinations of types, the arguments are automatically **coerced**
- this coercion may generate code (e.g. conversion from int to float)
- conversion is usually done towards more general types i.e. `10.5 + 0.5` has type `float` (since `float ≥ int`)

Coercion of Integer-types in C: Promotion

C defines special conversion rules for integers: **promotion**

```
unsigned char ≤ unsigned short ≤ int ≤ unsigned int
```

... where a conversion has to happen via all intermediate types.
Coercion of Integer-Types in C: Promotion

C defines special conversion rules for integers: promotion

\[
\begin{align*}
\text{unsigned char} & \leq \text{unsigned short} \\
\text{signed char} & \leq \text{signed short} \\
& \leq \text{int}\, \leq \text{unsigned int}
\end{align*}
\]

... where a conversion has to happen via all intermediate types.

subtle errors possible! Compute the character distribution of str:

```c
char* str = "...";
int dist[256];
memset(dist, 0, sizeof(dist));
while (*str) {
    dist[(*unsigned) +str ++] ++
    str ++
}
```

Note: unsigned is shorthand for unsigned int.

Subtypes

- on the arithmetic basic types char, int, long, etc. there exists a rich subtype hierarchy
- here \( t_1 \leq t_2 \), means that the values of type \( t_1 \)
  - form a subset of the values of type \( t_2 \);
  - can be converted into a value of type \( t_2 \);
  - fulfill the requirements of type \( t_2 \).

Example: assign smaller type (fewer values) to larger type

\[
\begin{align*}
t_1 & \ x; \\
t_2 & \ y; \\
y & = x;
\end{align*}
\]

Subtypes

- on the arithmetic basic types char, int, long, etc. there exists a rich subtype hierarchy
- here \( t_1 \leq t_2 \), means that the values of type \( t_1 \)
  - form a subset of the values of type \( t_2 \);
  - can be converted into a value of type \( t_2 \);
  - fulfill the requirements of type \( t_2 \).

Example: assign smaller type (fewer values) to larger type

\[
\begin{align*}
t_1 & \ x; \\
t_2 & \ y; \\
y & = x;
\end{align*}
\]

extend the subtype relationship to more complex types
Example: Subtyping

Observe:

```c
string extractInfo( struct { string info; } x) {  
    return x.info;
}
```

- we would like `extractInfo` to be applicable to all argument records that contain a field `string info`
- use deduction rules to describe when \( t_1 \leq t_2 \) should hold
- the idea of subtyping on values is related to subtyping as implemented in object-oriented languages

Rules for Well-Typedness of Subtyping

```
<table>
<thead>
<tr>
<th>t</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>A</td>
<td>t</td>
</tr>
<tr>
<td>A = s</td>
<td></td>
</tr>
</tbody>
</table>
```

```
struct {s_{j_1}, a_{j_1}; ... s_{j_m}, a_{j_m};} | struct {t_1, a_1; ... t_k, a_k;}
```

```
s_{j_1}, t_1
```

```
s_{j_m}, t_k
```

```
\ldots
```

```
struct {int u, int v}  x;
struct {int u}        y;
y = x;
```

Example: Subtyping

Observe:

```c
string extractInfo( struct { string info; } x) {  
    return x.info;
}
```

- we would like `extractInfo` to be applicable to all argument records that contain a field `string info`
- use deduction rules to describe when \( t_1 \leq t_2 \) should hold
- the idea of subtyping on values is related to subtyping as implemented in object-oriented languages

Rules and Examples for Subtyping

```
s_0 (s_1, \ldots, s_m) \quad t_0 (t_1, \ldots, t_m)
```

```
s_0 \quad t_0 \quad t_1 \quad s_1 \quad \ldots \quad t_m \quad s_m
```

Examples:

```
struct {int a; int b;}  struct {float a;}
int (int)               float (float)
int (float)             float (int)
```
Co- and Contra Variance

**Definition**
Given two function types in subtype relation \( s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n) \) then we have
- co-\textit{variance} of the return type \( s_0 \leq t_0 \) and
- contra-\textit{variance} of the arguments \( s_i \geq t_i \) für \( 1 < i \leq n \)

Example from functional languages:

\[ \text{int} \to \text{float} \rightarrow \text{int} \leq \text{int} \to \text{int} \rightarrow \text{float} \]

Subtypes: Application of Rules (I)

Check if \( S_1 \leq R_1 \):

- \( R_1 = \text{struct } \{ \text{int } a; R_1(R_1) f; \} \)
- \( S_1 = \text{struct } \{ \text{int } a; \text{int } b; S_1(S_1) f; \} \)
- \( R_2 = \text{struct } \{ \text{int } a; R_2(S_2) f; \} \)
- \( S_2 = \text{struct } \{ \text{int } a; \text{int } b; S_2(S_2) f; \} \)
Subtypes: Application of Rules (II)

Check if $S_2 \leq S_1$:

$$R_1 = \text{struct } \{ \text{int } a; \ R_1(R_2)\ f; \}$$
$$S_1 = \text{struct } \{ \text{int } a; \ \text{int } b; \ S_1(S_1)\ f; \}$$
$$R_2 = \text{struct } \{ \text{int } a; \ R_2(S_2)\ f; \}$$
$$S_2 = \text{struct } \{ \text{int } a; \ \text{int } b; \ S_2(R_2)\ f; \}$$

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

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
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 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 $\text{double}, \text{float}, \text{char}, \text{short}$ or $\text{long}$ 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 $\text{Dalvik VM}$ or the $\text{LLVM}$ 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 $\text{instructions}$
- instructions are executed on $\text{virtual hardware}$
- 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 $\text{run-time system}$, 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:

C

0 1

S

0

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 ~ stack.
- **SP**: (≡ 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

```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 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
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** 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

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

The Register Sets of the R-CMa
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>loadc $R_i$, $c$</td>
<td>$R_i = c$</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 $\rho x = a$):

\[
\begin{align*}
\text{code}^R_{\text{op}} e \rho &= \text{loadc } R_i, c \\
\text{code}^R_{\text{op}} x \rho &= \text{move } R_i, R_a \\
\text{code}^R_{\text{op}} x = e \rho &= \text{code}^R_{\text{op}} e \rho \\
\text{move } R_a, R_i \\
\end{align*}
\]

Note: all instructions use the Intel convention (in contrast to the AT&T convention): $\text{op } \text{dst} \ STC_1 \ldots \ STC_n$. 

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 $\text{op}$.

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

where $R_i$ is the target register, $R_j$ the first and $R_k$ the second argument.

Correspondingly, we generate code as follows:

\[
\begin{align*}
\text{code}^R_{\text{op}} e_1 \text{op } e_2 \rho &= \text{code}^R_{\text{op}} e_1 \rho \\
\text{code}^R_{\text{op}} e_2 \rho &= \text{code}^R_{\text{op}} e_1 \rho \\
\text{op } R_i, R_j, R_{k+1} \\
\end{align*}
\]
Translation of Expressions

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

\[
\text{code}_R^i \ e_1 \ \text{op} \ e_2 \ \rho = \begin{cases} 
\text{code}_R^{i+1} \ e_2 \ \rho \\
\text{op} \ R_i \ R_j \ R_{i+1}
\end{cases}
\]

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

\[
\text{code}_R^4 \ 3 \times 4 \ \rho = \text{code}_R^5 \ 3 \times 4 \ \rho \\
\text{add} \ R_4 \ R_3 \ R_5
\]

Managing Temporary Registers

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

\[
\text{code}_R^i \ 3 \times 4 + 3 \times 4 \ \rho = \begin{cases} 
\text{code}_R^i \ 3 \times 4 \ \rho \\
\text{add} \ R_4 \ R_3 \ R_5
\end{cases}
\]

where

\[
\text{code}_R^i \ 3 \times 4 \ \rho = \begin{cases} 
\text{load} \ R_3 \ 3 \\
\text{load} \ R_4 \ 4 \\
\text{mul} \ R_4 \ R_7 \ R_{i+1}
\end{cases}
\]

we obtain

\[
\text{code}_R^i \ 3 \times 4 + 3 \times 4 \ \rho = \begin{cases} 
\text{load} \ R_3 \ 3 \\
\text{load} \ R_4 \ 4 \\
\text{mul} \ R_4 \ R_3 \ R_5
\end{cases}
\]

Translation of Expressions

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

\[
\text{code}_R^i \ e_1 \ \text{op} \ e_2 \ \rho = \begin{cases} 
\text{code}_R^{i+1} \ e_2 \ \rho \\
\text{op} \ R_i \ R_j \ R_{i+1}
\end{cases}
\]

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

\[
\text{code}_R^4 \ 3 \times 4 \ \rho = \begin{cases} 
\text{load} \ R_4 \ 3 \\
\text{load} \ R_5 \ 4 \\
\text{mul} \ R_4 \ R_3 \ R_5
\end{cases}
\]

Managing Temporary Registers

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

\[
\text{code}_R^i \ 3 \times 4 + 3 \times 4 \ \rho = \begin{cases} 
\text{code}_R^i \ 3 \times 4 \ \rho \\
\text{add} \ R_4 \ R_3 \ R_5
\end{cases}
\]

where

\[
\text{code}_R^i \ 3 \times 4 \ \rho = \begin{cases} 
\text{load} \ R_3 \ 3 \\
\text{load} \ R_4 \ 4 \\
\text{mul} \ R_4 \ R_3 \ R_5
\end{cases}
\]

we obtain

\[
\text{code}_R^i \ 3 \times 4 + 3 \times 4 \ \rho = \begin{cases} 
\text{load} \ R_3 \ 3 \\
\text{load} \ R_4 \ 4 \\
\text{mul} \ R_4 \ R_5 \ R_5
\end{cases}
\]

\[
\text{add} \ R_4 \ R_4 \ R_5
\]
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`: $R_i = R_j * R_k$
- `mod R_i, R_j, R_k`: $R_i = \text{sgn}(R_k) \cdot \text{wobel}(R_j) = n|R_k| + k \wedge n \geq 0, 0 \leq k < |R_k|$
- `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`: $R_i = R_j \& R_k \quad // \text{bit-wise and}$
- `or R_i, R_j, R_k`: $R_i = R_j \mid R_k \quad // \text{bit-wise or}$

Translation of Unary Operators

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

```c
\text{code}_R^e \mathbf{op} e \rho = \text{code}_R^e \mathbf{op} R_i, R_i
```

Note: We use the same register.

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

```c
\text{code}_R^5 \boxed{\text{neg}} 4 \rho = \text{code}_R^5 \boxed{\text{neg}} R_5, R_5
```

Applying Translation Schema for Expressions

Suppose the following function is given:

```c
\text{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, $i = 4$.

```c
\text{code}_R^4 \boxed{\text{move}} R_4 R_4
```
Applying Translation Schema for Expressions

Suppose the following function

```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, $i = 4$.

```c
\begin{align*}
\text{code}_{i}^{4} x &= y + z * 3 \rho \\
&= \text{move } R_1 R_4 \\
\text{code}_{4}^{4} y + z * 3 \rho &= \text{move } R_4 R_2 \\
&= \text{add } R_4 R_4 R_5 \\
\text{code}_{4}^{5} z + 3 \rho &= \text{move } R_5 R_3 \\
&= \text{mul } R_0 R_5 R_6 \\
\text{code}_{0}^{6} 3 \rho &= \text{load } R_0 3
\end{align*}
```

The assignment $x = y + z * 3$ is translated as

```
move R_4 R_2; move R_5 R_3; load R_0 3; mul R_5 R_5 R_6; add R_4 R_4 R_5; move R_1 R_5
```

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

### 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}^{e} x = e \rho = \begin{cases} 
\text{code}_{i}^{e} e \rho & \text{move } R_a R_i
\end{cases}
```

However, $x = e$ is also an expression statement:
About Statements and Expressions

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

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

- Define:
  \[
  x = a \quad \Rightarrow \quad z = 42
  \]
  \[
  \text{code}^i e \_1 = e_2 \ \rho = \ \text{code}^i_{R_i} e_1 = e_2 \ \rho
  \]

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

Translation of Statement Sequences

The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:
\[
\begin{align*}
\text{code}^i_{(s \ s \ s)} & \ \rho = \ \text{code}^i_{s \ \rho} \\
\text{code}^i e & \ \rho = \ \text{code}^i_{s \ s \ \rho} \\
\# \ \text{empty sequence of instructions}
\end{align*}
\]
Note here: \( s \) is a statement, \( ss \) is a sequence of statements

Jumps

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

\[
\begin{align*}
\text{PC} & \quad \rightarrow \quad \text{jump A} \\
\text{PC} & \quad \rightarrow \quad \text{A}
\end{align*}
\]

Conditional Jumps

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

\[
\begin{align*}
\text{if} \ (R_i = 0) \ & \ \text{PC} = \ A; \\
\text{PC} & \quad \rightarrow \quad \text{jumpz Ri A} \\
\text{PC} & \quad \rightarrow \quad \text{A}
\end{align*}
\]
Management of Control Flow

In order to translate statements with control flow, we need to emit jump instructions.
- during the translation of an `if (c)` construct, it is not yet clear where to jump to in case that `c` is false

Management of Control Flow

In order to translate statements with control flow, we need to emit jump instructions.
- during the translation of an `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 unconditional jumps
  - minimize in a way so that fewer jumps are executed inside loops
  - replace far jumps through near jumps (if applicable)

To this end, we define:

**Definition**

A basic block consists of
- a sequence of statements `ss` that does not contain a jump
- a set of outgoing edges to other basic blocks
- where each edge may be labelled with a condition

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:
Formalizing the Translation Involving Control Flow
For simplicity of defining translations of instructions involving control flow, we use \textit{symbolic jump targets}.

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

Alternatively, we can emit \textit{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 (\(\sim\) near jumps)
- if all jumps are relative: the code becomes \textit{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 \textit{program optimization} where the statements inside basic blocks are simplified

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

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

\[
\text{code}^R \ s \rho = \text{code}^R \ c \rho \\
\text{jumpz} \ n \ A \\
\text{code}^R \ ss \rho
\]

Example for if-statement
Let \(\rho = \{ x \mapsto 4, y \mapsto 7 \}\) and let \(s\) be the statement

\[
\text{if} \ (x > y) \ { \\
\quad \text{*/ (i) */} \\
\quad x = x - y; \quad \text{*/ (ii) */} \\
\quad \text{else:} \\
\quad y = y - x; \quad \text{*/ (iii) */} \\
}\]

Then \(\text{code}^R \ s \rho\) yields:

\[
\begin{align*}
\text{move} \ R_i \ R_4 \\
\text{move} \ R_{i+1} \ R_7 \\
\text{gr} \ R_i \ R_{i+1} \\
\text{sub} \ R_i \ R_i \ R_{i+1} \\
\text{move} \ R_{i+1} \ R_i \\
\text{jumpz} \ R_i \ A \\
\text{jump} \ B \\
\end{align*}
\]
Example for if-statement

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

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

Then $\text{code}^i s \rho$ yields:

1. move $R_i R_4$
2. move $R_i R_4 R_7$
3. sub $R_i R_i R_{i+1}$
4. jumpz $R_i A$

for-Loops

The for-loop $s \equiv \text{for} (e_1; e_2; e_3) s'$ is equivalent to the statement sequence $e_1$; while $(e_2) \{s' e_3;\} - \text{as long as } s' \text{ does not contain a continue statement.}$

Thus, we translate:

```
\text{code}^i \text{for} (e_1; e_2; e_3) s \rho = \text{code}^{i_1} e_1 \rho \\
A: \quad \text{code}^{i_2} e_2 \rho \\
\quad \text{jumpz} R_i B \\
\quad \text{code}^{i_3} s \rho \\
\quad \text{jump} A
```

Iterating Statements

We only consider the loop $s \equiv \textbf{while} (e) s'$. For this statement we define:

```
\text{code}^i \textbf{while} (e) s \rho = A: \quad \text{code}^{i_1} e \rho \\
\quad \text{jump} R_i, B \\
\text{code}^{i_3} s \rho \\
B: \quad \text{jump} A
```

The switch-Statement

**Idea:**
- Suppose choosing from multiple options in constant time if possible
- use a \textit{jump table} that, at the i\textsuperscript{th} position, holds a jump to the i\textsuperscript{th} alternative
- in order to realize this idea, we need an \textit{indirect jump} instruction