Topic:

Code Synthesis
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
Chapter 1:
The Register C-Machine
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 \textbf{double}, \textbf{float}, \textbf{char}, \textbf{short} or \textbf{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 \textit{Dalvik VM/ART} or the \textit{LLVM} are similar to the R-CMa
- an interpreter of R-CMa can run on any platform
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; stack.
- **SP**: (pronounced stack pointer) pointer to the last used cell in S;
- beyond S follows the memory containing the heap;
- **C** is the memory storing code:
  - each cell of C holds exactly one virtual instruction;
  - C can only be read;
- **PC** (pronounced program counter) address of the instruction that is to be executed next;
- PC contains 0 initially.
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

```c
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
Chapter 2:
Generating Code for the Register C-Machine
Task: evaluate the expression \((1 + 7) \times 3\)
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:

1. 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

2. 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><code>loadc Ri c</code></td>
<td>$R_i = c$</td>
<td>load constant</td>
</tr>
<tr>
<td><code>move Ri Rj</code></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 $\rho x = a$):

$$
\begin{align*}
\text{code}^i_R c \rho &= \text{loadc} R_i c \\
\text{code}^i_R x \rho &= \text{move} R_i R_a \\
\text{code}^i_R x = e \rho &= \text{code}^i_R e \rho \\
\text{code}^i_R x = e \rho &= \text{move} R_a R_i
\end{align*}
$$
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 \( \text{op} \).

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

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

Correspondingly, we generate code as follows:

\[
\text{code}_i^e \quad e_1 \quad \text{op} \quad e_2 \quad \rho = \quad \text{code}_i^e \quad e_1 \quad \rho \\
\text{code}_{i+1}^e \quad e_2 \quad \rho \\
\text{op} \quad R_i \quad R_i \quad R_{i+1}
\]

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

\[
\text{code}_4^3 \quad 3 \quad \times \quad 4 \quad \rho = \quad \text{code}_4^3 \quad 3 \quad \rho \\
\text{code}_5^4 \quad 4 \quad \rho \\
\text{loadc} \quad R_4 \quad 3 \\
\text{loadc} \quad R_5 \quad 4 \\
\text{mul} \quad R_4 \quad R_4 \quad R_5
\]
Managing Temporary Registers

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

$$\text{code}_R^4 \ 3 \times 4 + 3 \times 4 \ \rho = \text{code}_R^4 \ 3 \times 4 \ \rho$$

$$\text{code}_R^4 \ 3 \times 4 \ \rho = \text{code}_R^5 \ 3 \times 4 \ \rho$$

$$\text{add} \ R_4 \ R_4 \ R_5$$

where

$$\text{code}_R^i \ 3 \times 4 \ \rho = \text{loadc} \ R_i \ 3$$

$$\text{loadc} \ R_{i+1} \ 4$$

$$\text{mul} \ R_i \ R_i \ R_{i+1}$$

we obtain

$$\text{code}_R^4 \ 3 \times 4 + 3 \times 4 \ \rho = \text{loadc} \ R_4 \ 3$$

$$\text{loadc} \ R_5 \ 4$$

$$\text{mul} \ R_4 \ R_4 \ R_5$$

$$\text{loadc} \ R_5 \ 3$$

$$\text{loadc} \ R_6 \ 4$$

$$\text{mul} \ R_5 \ R_5 \ R_6$$

$$\text{add} \ R_4 \ R_4 \ R_5$$
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** \( R_i \) \( R_j \) \( R_k \) \( R_i = R_j \times R_k \)
- **mod** \( R_i \) \( R_j \) \( R_k \) \( R_i = \text{signum}(R_k) \times k \) with \( |R_j| = n \cdot |R_k| + k \land n \geq 0, 0 \leq k < |R_k| \)
- **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** \( R_i \) \( R_j \) \( R_k \) \( R_i = R_j \& R_k \) // bit-wise and
- **or** \( R_i \) \( R_j \) \( R_k \) \( R_i = R_j \mid R_k \) // bit-wise or

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

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

$$\text{code}_R^i \ \text{op} \ e \ \rho = \ \text{code}_R^i \ e \ \rho \ \text{op} \ R_i \ R_i$$

Note: We use the same register.

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

$$\text{code}_R^5 \ -4 \ \rho = \ \text{code}_R^5 \ 4 \ \rho$$
$$\text{code}_R^5 \ -4 \ \rho = \ \text{loadc} \ R_5 \ 4 \ \neg R_5 \ R_5$$

The operators have the following semantics:

$$\text{not} \ R_i \ R_j \quad R_i \leftarrow \text{if } R_j = 0 \text{ then } 1 \text{ else } 0$$
$$\text{neg} \ R_i \ R_j \quad R_i \leftarrow -R_j$$
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, \( i = 4 \).

\[
\begin{align*}
\text{code}^4 & \quad x = y + z \times 3 \quad \rho &= \text{code}^4_R \quad y + z \times 3 \quad \rho \\
& \quad \text{move} \quad R_1 \quad R_4 \\
\text{code}^4 & \quad y + z \times 3 \quad \rho &= \text{move} \quad R_4 \quad R_2 \\
& \quad \text{code}^5_R \quad z \times 3 \quad \rho \\
& \quad \text{add} \quad R_4 \quad R_4 \quad R_5 \\
\text{code}^5 & \quad z \times 3 \quad \rho &= \text{move} \quad R_5 \quad R_3 \\
& \quad \text{code}^6_R \quad 3 \quad \rho \\
& \quad \text{mul} \quad R_5 \quad R_5 \quad R_6 \\
\text{code}^6 & \quad 3 \quad \rho &= \text{loadc} \quad R_6 \quad 3
\end{align*}
\]

\( \mapsto \) the assignment \( x = y + z \times 3 \) is translated as

```c
move R_4 R_2; move R_5 R_3; loadc R_6 3; mul R_5 R_5 R_6; add R_4 R_4 R_5; move R_1 R_4
```
Chapter 3:
Statements and Control Structures
About Statements and Expressions

General idea for translation:
- \( \text{code}^i s \rho \): generate code for statement \( s \)
- \( \text{code}^i_{R} e \rho \): generate code for expression \( e \) into \( R_i \)

Throughout: \( i, i + 1, \ldots \) are free (unused) registers

For an expression \( x = e \) with \( \rho x = a \) we defined:

\[
\text{code}^i_{R} x = e \rho = \text{code}^i_{R} e \rho
\]

move \( R_a \ R_i \)

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

- Define:

\[
\text{code}^i e_1 = e_2; \ \rho = \text{code}^i_{R} e_1 = e_2 \ \rho
\]

The temporary register \( R_i \) is ignored here. More general:

\[
\text{code}^i e; \ \rho = \text{code}^i_{R} e \ \rho
\]

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

The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

$$\text{code}^i \ (s\ ss) \ \rho = \ \text{code}^i \ s \ \rho$$

$$\text{code}^i \ ss \ \rho$$

$$\text{code}^i \ \varepsilon \ \rho = \ // \text{ empty sequence of instructions}$$

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:

\[
\text{PC} = A;
\]
A conditional jump branches depending on the value in $R_i$:

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

We first consider $s \equiv \textbf{if } (c) \textbf{ ss}$. 
...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

\[
\text{code}_i^s \rho = \text{code}_R^c \rho 
\]
\[
\text{jumpz } R_i A \\
\text{code}_i^ss \rho \\
A: \ldots
\]
Translation of $\textbf{if } (c) \textbf{ tt else ee}$. 

$$\text{code}^i \textbf{if}(c) \textbf{ tt else ee } \rho =
\begin{align*}
\text{code}_R^i & \ c \rho \\
\text{jumpz} & \\
\text{jumpz } R_i & A \\
\text{code}^i & \textbf{ tt } \rho \\
\text{jump} & \\
A : & \text{code}^i \textbf{ ee } \rho \\
B : & \\
\end{align*}$$
Example for if-statement

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

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

Then code $s \rho$ yields:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(ii)</th>
<th>(iii)</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $R_i$ $R_4$</td>
<td>move $R_i$ $R_4$</td>
<td>A : move $R_i$ $R_7$</td>
</tr>
<tr>
<td>move $R_{i+1}$ $R_7$</td>
<td>move $R_{i+1}$ $R_7$</td>
<td>move $R_{i+1}$ $R_4$</td>
</tr>
<tr>
<td>gr $R_i$ $R_i$ $R_{i+1}$</td>
<td>sub $R_i$ $R_i$ $R_{i+1}$</td>
<td>sub $R_i$ $R_i$ $R_{i+1}$</td>
</tr>
<tr>
<td>jumpz $R_i$ $A$</td>
<td>move $R_4$ $R_i$</td>
<td>move $R_7$ $R_i$</td>
</tr>
<tr>
<td>jump $B$</td>
<td>B :</td>
<td></td>
</tr>
</tbody>
</table>

25 / 49
We only consider the loop $s \equiv \textbf{while} \ (e) \ s'$. For this statement we define:

$$\text{code}^i \textbf{while}(e) \ s \ \rho = A : \ \text{code}^i_R \ e \ \rho$$

$$\begin{align*}
\text{jumpz} \ R_i \ B \\
\text{code}^i \ s \ \rho \\
\text{jump} \ A
\end{align*}$$

$B :$
Example: Translation of Loops

Let \( \rho = \{ a \mapsto 7, b \mapsto 8, c \mapsto 9 \} \) and let \( s \) be the statement:

```cpp
while (a > 0) {
    /* (i) */
    c = c + 1;
    /* (ii) */
    a = a - b;
    /* (iii) */
}
```

Then code \( s \rho \) evaluates to:

(i) \[ A : \]
- move \( R_i R_7 \)
- loadc \( R_{i+1} 0 \)
- gr \( R_i R_i R_{i+1} \)
- jumpz \( R_i B \)

(ii) \[ (ii) \]
- move \( R_i R_9 \)
- loadc \( R_{i+1} 1 \)
- add \( R_i R_i R_{i+1} \)
- move \( R_9 R_i \)
- move \( R_7 R_i \)
- jump \( A \)

B :
for-Loops

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

Thus, we translate:

$$\text{code}^i \text{ for}(e_1; e_2; e_3) s \rho = \text{code}^i e_1 \rho$$

$$A : \text{code}^i e_2 \rho$$

$$\text{jumpz } R_i B$$

$$\text{code}^i s \rho$$

$$\text{code}^i e_3 \rho$$

$$\text{jump } A$$

$$B :$$
The switch-Statement

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

\[
\text{PC} = A + R_i;
\]
Consecutive Alternatives

Let \texttt{switch} \( s \) be given with \( k \) consecutive \texttt{case} alternatives:

\begin{verbatim}
switch (e) {
    case 0: s0; break;
    ... 
    case k - 1: sk-1; break;
    default: sk; break;
}
\end{verbatim}

Define \( \text{code}^i s \rho \) as follows:

\begin{align*}
\text{code}^i s \rho &= \text{code}^i e \rho \\
\text{check}^i 0 k B &= B : \text{jump A0} \\
A0 : \text{code}^i s0 \rho &= \text{jump C} \\
\vdots & \vdots \\
Ak : \text{code}^i sk \rho &= \text{jump Ak}
\end{align*}

\( \text{check}^i l u B \) checks if \( l \leq R_i < u \) holds and jumps accordingly.
Translation of the $\text{check}^i$ Macro

The macro $\text{check}^i l u B$ checks if $l \leq R_i < u$. Let $k = u - l$.

- If $l \leq R_i < u$ it jumps to $B + R_i - l$
- If $R_i < l$ or $R_i \geq u$ it jumps to $A_k$

we define:

\[
\text{check}^i l u B = \text{loadc } R_{i+1} l \\
\text{geq } R_{i+2} R_i R_{i+1} \\
\text{jumpz } R_{i+2} E \\
\text{sub } R_i R_i R_{i+1} \\
\text{loadc } R_{i+1} k \\
\text{geq } R_{i+2} R_i R_{i+1} \\
\text{jumpz } R_{i+2} D \\
E: \text{ loadc } R_i k \\
D: \text{ jumpi } R_i B
\]

Note: a jump $\text{jumpi } R_i B$ with $R_i = u$ winds up at $B + u$, the default case
Improvements for Jump Tables

This translation is only suitable for certain switch-statement.

- In case the table starts with 0 instead of \( u \) we don’t need to subtract it from \( e \) before we use it as index.
- If the value of \( e \) is guaranteed to be in the interval \([l, u]\), we can omit check.
General translation of switch-Statements

In general, the values of the various cases may be far apart:

- generate an if-ladder, that is, a sequence of if-statements
- for $n$ cases, an if-cascade (tree of conditionals) can be generated $\sim O(\log n)$ tests
- if the sequence of numbers has small gaps ($\leq 3$), a jump table may be smaller and faster
- one could generate several jump tables, one for each sets of consecutive cases
- an if cascade can be re-arranged by using information from profiling, so that paths executed more frequently require fewer tests
Chapter 4: Functions
Ingredients of a Function

The definition of a function consists of

- a name with which it can be called;
- a specification of its formal parameters;
- possibly a result type;
- a sequence of statements.

In C we have:

\[
\text{code}_R^i f \rho = \text{loadc } R_i \_f \quad \text{with } \_f \text{ starting address of } f
\]

Observe:

- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later
Memory Management in Functions

```c
int fac(int x) {
    if (x<=0) return 1;
    else return x*fac(x-1);
}

int main(void) {
    int n;
    n = fac(2) + fac(1);
    printf("%d", n);
}
```

At run-time several instances may be active, that is, the function has been called but has not yet returned.
The recursion tree in the example:

```
main
   fac
   fac
   printf
   fac
   fac
   fac
```
The **formal parameters** and the **local variables** of the various **instances** of a function must be kept separate

**Idea for implementing functions:**

- set up a region of memory each time it is called
- in sequential programs this memory region can be allocated on the stack
- thus, each instance of a function has its own region on the stack
- these regions are called **stack frames**
Organization of a Stack Frame

- **stack** representation: grows upwards
- **SP** points to the last used stack cell

\[ \text{FP} \equiv \text{frame pointer}: \text{points to the last organizational cell} \]

- **FP** points to the last organizational cell used to recover the previously active stack frame

![Diagram of stack frame organization](image-url)
Definition

Let $f$ be the current function that calls a function $g$.
- $f$ is dubbed *caller*
- $g$ is dubbed *callee*

The code for managing function calls has to be split between caller and callee. This split cannot be done arbitrarily since some information is only known in that caller or only in the callee.

Observation:

The space requirement for parameters is only known by the caller:
Example: `printf`
Principle of Function Call and Return

actions taken on entering \( g \):

1. compute the start address of \( g \)
2. compute actual parameters in globals
3. backup of caller-save registers 
4. backup of FP
5. set the new FP
6. back up of PC and 
   jump to the beginning of \( g \)
7. copy actual params to locals 

\[ \{ \text{saveloc} \} \text{mark} \{ \text{call} \} \{ \text{return} \} \{ \text{restoreloc} \} \text{are in } f \]

actions taken on leaving \( g \):

1. compute the result into \( R_0 \)
2. restore FP, SP
3. return to the call site in \( f \), 
   that is, restore PC
4. restore the caller-save registers

\[ \{ \text{return} \} \{ \text{restoreloc} \} \text{is in } f \]
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in \textit{local} registers \( R_i \)
- intermediate results also live in \textit{local} registers \( R_i \)
- parameters live in \textit{global} registers \( R_i \) (with \( i \leq 0 \))
- global variables: let’s suppose there are none

convention:

- the \( i \)th argument of a function is passed in register \( R_{-i} \)
- the result of a function is stored in \( R_0 \)
- local registers are saved before calling a function

\textbf{Definition}

Let \( f \) be a function that calls \( g \). A register \( R_i \) is called

- \textit{caller-saved} if \( f \) backs up \( R_i \) and \( g \) may overwrite it
- \textit{callee-saved} if \( f \) does not back up \( R_i \), and \( g \) must restore it before returning
Translation of Function Calls

A function call $g(e_1, \ldots e_n)$ is translated as follows:

$$\text{code}_R^i \ g(e_1, \ldots e_n) \ \rho = \ \text{code}_R^i \ g \ \rho$$

$$\text{code}_R^{i+1} \ e_1 \ \rho$$

$$\vdots$$

$$\text{code}_R^{i+n} \ e_n \ \rho$$

move $R_{-1} R_{i+1}$

$$\vdots$$

move $R_{-n} R_{i+n}$

saveloc $R_1 R_{i-1}$

mark

call $R_i$

restoreloc $R_1 R_{i-1}$

New instructions:

- saveloc $R_i R_j$ pushes the registers $R_i, R_{i+1} \ldots R_j$ onto the stack
- mark backs up the organizational cells
- call $R_i$ calls the function at the address in $R_i$
- restoreloc $R_i R_j$ pops $R_j, R_{j-1}, \ldots R_i$ off the stack
Rescuing the FP

The instruction \textit{mark} allocates stack space for the return value and the organizational cells and backs up FP.

\begin{align*}
\text{S}[\text{SP}+1] &= \text{FP}; \\
\text{SP} &= \text{SP} + 1;
\end{align*}
Calling a Function

The instruction `call` rescues the value of PC+1 onto the stack and sets FP and PC.

```
SP = SP+1;
S[SP] = PC;
FP = SP;
PC = Ri;
```
Result of a Function

The global register set is also used to communicate the result value of a function:

\[
\text{code}^i \ 	ext{return} \ e \ \rho \ = \ \text{code}^i_R \ e \ \rho \\
\text{move} \ R_0 \ R_i \\
\text{return}
\]

alternative without result value:

\[
\text{code}^i \ 	ext{return} \ \rho \ = \ \text{return}
\]

*global* registers are otherwise not used inside a function body:

- advantage: at any point in the body another function can be called without backing up *global* registers
- disadvantage: on entering a function, all *global* registers must be saved
Return from a Function

The instruction `return` relinquishes control of the current stack frame, that is, it restores `PC` and `FP`.

PC = S[FP];
SP = FP-2;
FP = S[SP+1];
Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^1 t_r \ f(\text{args})\{\text{decls} \ ss\} \ \rho = \ \text{move} \ R_{l+1} \ R_{-1} \\
\vdots \\
\text{move} \ R_{l+n} \ R_{-n} \\
\text{code}^{l+n+1} \ ss \ \rho' \\
\text{return}
\]

Assumptions:

- the function has \(n\) parameters
- the local variables are stored in registers \(R_1, \ldots, R_l\)
- the parameters of the function are in \(R_{-1}, \ldots, R_{-n}\)
- \(\rho'\) is obtained by extending \(\rho\) with the bindings in \(\text{decls}\) and the function parameters \(\text{args}\)
- \text{return} is not always necessary

Are the \text{move} instructions always necessary?
Translation of Whole Programs

A program $P = F_1; \ldots F_n$ must have a single main function.

\[
\text{code}^1 P \rho = \text{loadc } R_1 \_\text{main} \\
\text{mark} \\
\text{call } R_1 \\
\text{halt} \\
_f 1: \text{code}^1 F_1 \rho \oplus \rho_{f_1} \\
\vdots \\
_f n: \text{code}^1 F_n \rho \oplus \rho_{f_n}
\]

Assumptions:
- $\rho = \emptyset$ assuming that we have no global variables
- $\rho_{f_i}$ contain the addresses of the functions up to $f_i$
- $\rho_1 \oplus \rho_2 = \lambda x . \begin{cases} 
\rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\
\rho_1(x) & \text{otherwise}
\end{cases}$
Translation of the \texttt{fac}-function

Consider:

\begin{verbatim}
int fac(int x) {
    if (x<=0)
        return 1;
    else
        return x*fac(x-1);
}
\end{verbatim}

\begin{verbatim}
_\texttt{fac}:
  move R1 R-1  \text{save param.}
  i = 2  move R2 R1  \text{if (x<=0)}
  loadc R3 0
  leq R2 R2 R3
  jumpz R2 _A
  loadc R1 1
  move R0 R2
  return
  jump _B    \text{code is dead}

_\texttt{A}:
  move R2 R1
  i = 3  loadc R3 _\texttt{fac}
  i = 4  move R4 R1
  i = 5  loadc R5 1
  i = 6  sub R4 R4 R5
  i = 5  move R-1 R4
  i = 3  saveloc R1 R2
          \text{mark}
          \text{call R3}
          \text{restoreloc R1 R2}
          \text{move R3 R0}
  i = 4  mul R2 R2 R3
  i = 3  move R0 R2
          return
          return

_\texttt{B}:
  return
\end{verbatim}