**Script** generated by TTT

| | |
|---|---|
| Title: | Petter: Compilerbau (04.07.2016) |
| Date: | Mon Jul 04 14:33:06 CEST 2016 |
| Duration: | 88:02 min |
| Pages: | 42 |

## 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

$\leadsto$ we commence by specifying machine instruction semantics

# Chapter 1:
# The Register C-Machine

## The Register C-Machine (R-CMa)

VAM

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 $\boxed{\text{~~R-CMa~~}}$
- 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 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

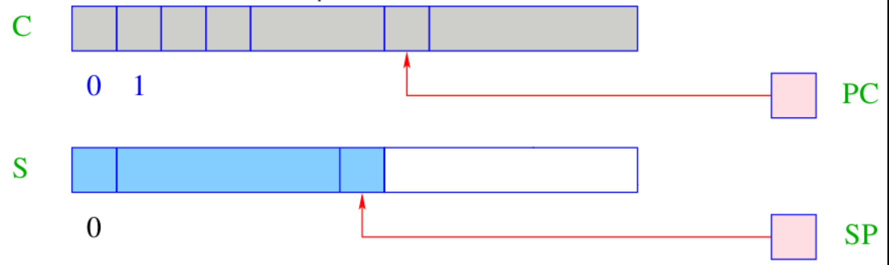
## Virtual Machines

A virtual machine has the following ingredients:

- any virtual machine provides a set of $\boxed{\text{instructions}}$
- instructions are executed on $\boxed{\text{virtual hardware}}$
- 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 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:

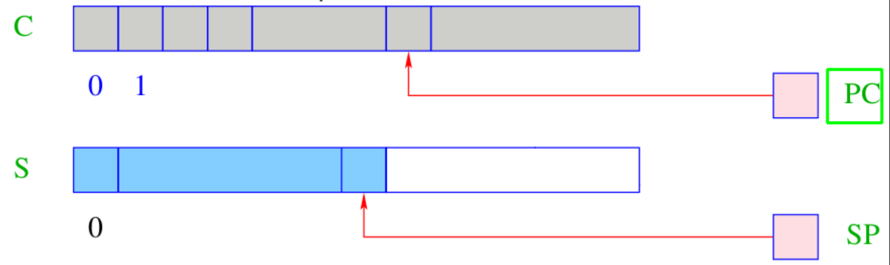


A virtual machine such as the Dalvik VM has the following structure:

- S: the data store – a memory region in which cells can be stored in LIFO order $\leadsto$ stack.
- SP: ($\widehat{=}$ stack pointer) pointer to the last used cell in S
- beyond S follows the memory containing the heap

## 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 $\leadsto$ stack.
- SP: ($\widehat{=}$ 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 ($\widehat{=}$ 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

```
          flag
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

## Simple Expressions and Assignments in R-CMa

Task: evaluate the expression $\boxed{(1+7)*3}$
that is, generate an instruction sequence that
- computes the value of the expression and
- keeps its value accessible in a reproducable way

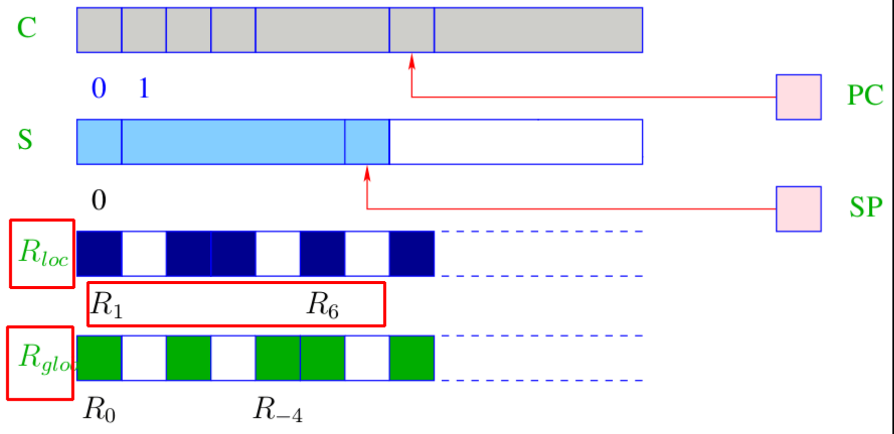
## Simple Expressions and Assignments in R-CMa

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 reproducable 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 $R_1, R_2, \dots R_i, \dots$
- *global* register are $R_0, R_{-1}, \dots R_j, \dots$



## 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

## 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 $\boxed{\text{register counter } i\text{:}}$

- 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:

| instruction | semantics | intuition |
|---|---|---|
| $\boxed{\text{loadc } R_i \boxed{c}}$ | $R_i = c$ | load constant |
| $\boxed{\text{move } R_i \boxed{R_j}}$ | $R_i = R_j$ | copy $R_j$ to $R_i$ |

instr dst $src_1$ ... $src_n$

## Translation of Simple Expressions

Using variables stored in registers; loading constants:

| instruction | semantics | intuition |
|---|---|---|
| loadc $R_i$ $c$ | $R_i = c$ | load constant |
| move $R_i$ $R_j$ | $R_i = R_j$ | copy $R_j$ to $R_i$ |

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

$$
\begin{aligned}
\text{code}_R^i\ c\ \rho &= \text{loadc } R_i\ c \\
\text{code}_R^i\ x\ \rho &= \text{move } R_i\ R_a \\
\text{code}_R^i\ x = e\ \rho &= \text{code}_R^i\ e\ \rho \\
&\quad \text{move } R_a\ R_i
\end{aligned}
$$

## Translation of Expressions

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

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:

$$
\begin{aligned}
\text{code}_R^i\ e_1\ \text{op}\ e_2\ \rho = &\ \text{code}_R^i\ e_1\ \rho \\
&\ \text{code}_R^{i+1}\ e_2\ \rho \\
&\ \text{op}\ R_i\ R_i\ R_{i+1}
\end{aligned}
$$

## Translation of Expressions

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

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:

$$
\begin{aligned}
\text{code}_R^i\ e_1\ \text{op}\ e_2\ \rho = &\ \text{code}_R^i\ e_1\ \rho \\
&\ \text{code}_R^{i+1}\ e_2\ \rho \\
&\ \text{op}\ R_i\ R_i\ R_{i+1}
\end{aligned}
$$

Example: Translate $\boxed{3 * 4}$ with $\boxed{i = 4}$:

$$
\begin{aligned}
\text{code}_R^{\boxed{4}}\ \boxed{3*4}\ \rho = &\ \text{code}_R^{\boxed{4}}\ \boxed{3}\ \rho \\
&\ \text{code}_R^{\boxed{5}}\ \boxed{4}\ \rho \\
&\ \boxed{\text{mul}\ R_4\ \boxed{R_4}\ \boxed{R_5}}
\end{aligned}
$$

## Translation of Expressions

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

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:

$$
\begin{aligned}
\text{code}_R^i\ e_1\ \text{op}\ e_2\ \rho = &\ \text{code}_R^i\ e_1\ \rho \\
&\ \text{code}_R^{i+1}\ e_2\ \rho \\
&\ \text{op}\ R_i\ R_i\ R_{i+1}
\end{aligned}
$$

Example: Translate $3 * 4$ with $i = 4$:

$$
\begin{aligned}
\text{code}_R^4\ 3*4\ \rho = &\ \text{loadc}\ R_4\ 3 \\
&\ \text{loadc}\ R_5\ 4 \\
&\ \text{mul}\ R_4\ R_4\ R_5
\end{aligned}
$$

## Managing Temporary Registers

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

$$\mathrm{code}^4_\mathrm{R}\ \ 3{*}4{+}3{*}4\ \ \rho \quad = \quad \begin{array}{l} \mathrm{code}^4_\mathrm{R}\ \ 3{*}4\ \ \rho \\ \mathrm{code}^5_\mathrm{R}\ \ 3{*}4\ \ \rho \\ \mathrm{add}\ R_4\ R_4\ R_5 \end{array}$$

where

$$\mathrm{code}^i_\mathrm{R}\ \ 3{*}4\ \ \rho \quad = \quad \begin{array}{l} \mathrm{loadc}\ R_i\ 3 \\ \mathrm{loadc}\ R_{i+1}\ 4 \\ \mathrm{mul}\ R_i\ R_i\ R_{i+1} \end{array}$$

we obtain

$$\mathrm{code}^4_\mathrm{R}\ \ 3{*}4{+}3{*}4\ \ \rho \quad =$$

## Managing Temporary Registers

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

$$\mathrm{code}^4_\mathrm{R}\ \ 3{*}4{+}3{*}4\ \ \rho \quad = \quad \begin{array}{l} \mathrm{code}^4_\mathrm{R}\ \ 3{*}4\ \ \rho \\ \mathrm{code}^5_\mathrm{R}\ \ 3{*}4\ \ \rho \\ \mathrm{add}\ R_4\ R_4\ R_5 \end{array}$$

where

$$\mathrm{code}^i_\mathrm{R}\ \ 3{*}4\ \ \rho \quad = \quad \begin{array}{l} \mathrm{loadc}\ R_i\ 3 \\ \mathrm{loadc}\ R_{i+1}\ 4 \\ \mathrm{mul}\ R_i\ R_i\ R_{i+1} \end{array}$$

we obtain

$$\mathrm{code}^4_\mathrm{R}\ \ 3{*}4{+}3{*}4\ \ \rho \quad = \quad \begin{array}{l} \mathrm{loadc}\ R_4\ 3 \\ \mathrm{loadc}\ R_5\ 4 \\ \mathrm{mul}\ R_4\ R_4\ R_5 \\ \mathrm{loadc}\ R_5\ 3 \\ \mathrm{loadc}\ R_6\ 4 \\ \mathrm{mul}\ R_5\ R_5\ R_6 \\ \mathrm{add}\ R_4\ R_4\ R_5 \end{array}$$

## 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 $R_i\ R_j\ R_k$ | $R_i = \mathit{signum}(R_k) \cdot k$ with $\lvert R_j \rvert = n \cdot \lvert R_k \rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k \rvert$ |
| le $R_i\ R_j\ R_k$ | $R_i =$ if $R_j < R_k$ then $1$ else $0$ |
| gr $R_i\ R_j\ R_k$ | $R_i =$ if $R_j > R_k$ then $1$ else $0$ |
| eq $R_i\ R_j\ R_k$ | $R_i =$ if $R_j = R_k$ then $1$ else $0$ |
| leq $R_i\ R_j\ R_k$ | $R_i =$ if $R_j \leq R_k$ then $1$ else $0$ |
| geq $R_i\ R_j\ R_k$ | $R_i =$ if $R_j \geq R_k$ then $1$ 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 op $= \{\mathit{neg}, \mathit{not}\}$ take only two registers:

$$\mathrm{code}^i_\mathrm{R}\ \boxed{\mathrm{op}\ e}\ \rho \quad = \quad \begin{array}{l} \boxed{\mathrm{code}^i_\mathrm{R}}\ e\ \rho \\ \boxed{\mathrm{op}}\ \boxed{R_i}\ \boxed{R_i} \end{array}$$

## Translation of Unary Operators

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

$$\begin{array}{lcl} \text{code}^i_{\text{R}}\ \text{op}\ e\ \rho & = & \text{code}^i_{\text{R}}\ e\ \rho \\ & & \text{op}\ R_i\ R_i \end{array}$$

Note: We use the same register.

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

$$\begin{array}{lcl} \text{code}^5_{\text{R}}\ -4\ \rho & = & \boxed{\text{code}^5_{\text{R}}\ 4}\ \rho \\ & & \boxed{\text{neg}}\ R_5\ R_5 \end{array}$$

## Translation of Unary Operators

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

$$\begin{array}{lcl} \text{code}^i_{\text{R}}\ \text{op}\ e\ \rho & = & \text{code}^i_{\text{R}}\ e\ \rho \\ & & \text{op}\ R_i\ R_i \end{array}$$

Note: We use the same register.

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

$$\begin{array}{lcl} \text{code}^5_{\text{R}}\ -4\ \rho & = & \text{loadc}\ R_5\ 4 \\ & & \text{neg}\ R_5\ R_5 \end{array}$$

The operators have the following semantics:

$$\begin{array}{ll} \text{not}\ R_i\ R_j & R_i \leftarrow \text{if}\ \boxed{R_j = 0}\ \text{then}\ \boxed{1}\ \text{else}\ \boxed{0} \\ \text{neg}\ R_i\ R_j & R_i \leftarrow \boxed{-R_j} \end{array}$$

## Applying Translation Schema for Expressions

Suppose the following function is given:

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

Let $\rho = \{\boxed{x \mapsto 1}, \boxed{y \mapsto 2}, \boxed{z \mapsto 3}\}$ be the address environment.

Let $\boxed{R_4}$ be the first free register, that is, $\boxed{i = 4}$.

$$\text{Code}^i_{\text{R}}\ \rho \qquad \text{code}^{\boxed{4}}_{\text{R}}\ \texttt{x=y+z*3}\ \rho \quad = \quad \boxed{\text{code}^4_{\text{R}}\ \texttt{y+z*3}\ \rho}$$
$$\boxed{\text{move}\ R_1\ R_4}$$

## Applying Translation Schema for Expressions

Suppose the following function is given:

```
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, $i = 4$.

$$\begin{array}{lcl} \text{code}^4\ \texttt{x=y+z*3}\ \rho & = & \boxed{\text{code}^4_{\text{R}}\ \texttt{y+z*3}\ \rho} \\ & & \text{move}\ R_1\ R_4 \\ \boxed{\text{code}^4_{\text{R}}\ \boxed{\texttt{y}}\texttt{+z*3}\ \rho} & = & \text{move}\ \boxed{R_4}\ \boxed{R_2} \\ & & \text{code}^5_{\text{R}}\ \texttt{z*3}\ \rho \\ & & \boxed{\text{add}\ R_4\ \boxed{R_4}\ \boxed{R_5}} \end{array}$$

## Applying Translation Schema for Expressions

Suppose the following function is given:

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

Let $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto \boxed{3}\}$ be the address environment.

Let $R_4$ be the first free register, that is, $i = 4$.

$$\begin{aligned}
\text{code}^4\ \texttt{x=y+z*3}\ \rho &= \text{code}_R^4\ \texttt{y+z*3}\ \rho \\
&\quad \text{move}\ R_1\ R_4 \\
\text{code}_R^4\ \texttt{y+z*3}\ \rho &= \text{move}\ R_4\ R_2 \\
&\quad \boxed{\text{code}_R^{\boxed{5}}\ \texttt{z*3}\ \rho} \\
&\quad \text{add}\ R_4\ R_4\ R_5 \\
\boxed{\text{code}_R^{\boxed{5}}\ \boxed{\texttt{z*}}\texttt{3}\ \rho} &= \text{move}\ \boxed{R_5}\ \boxed{R_3} \\
&\quad \text{code}_R^6\ 3\ \rho \\
&\quad \boxed{\text{mul}\ R_5\ R_5\ R_6}
\end{aligned}$$

## Applying Translation Schema for Expressions

Suppose the following function is given:

```
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, $i = 4$.

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

## Applying Translation Schema for Expressions

Suppose the following function is given:

```
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, $i = 4$.

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

$\leadsto$ the assignment `x=y+z*3` is translated as

$\text{move}\ R_4\ R_2;\ \text{move}\ R_5\ R_3;\ \text{loadc}\ R_6\ 3;\ \text{mul}\ R_5\ R_5\ R_6;\ \text{add}\ R_4\ R_4\ R_5;\ \text{move}\ R_1\ R_4$

# Chapter 3:

# Statements and Control Structures

## About Statements and Expressions

General idea for translation:

$\text{code}^i\ s\ \rho$ : generate code for statement $s$

$\text{code}^i_{\text{R}}\ e\ \rho$ : generate code for expression $e$ into $R_i$

Throughout: $\boxed{i, i+1, \ldots}$ are free (unused) registers

## About Statements and Expressions

General idea for translation:

$\text{code}^i\ s\ \rho$ : generate code for statement $s$

$\text{code}^i_{\text{R}}\ e\ \rho$ : generate code for expression $e$ into $R_i$

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

For an *expression* $\boxed{x = e}$ with $\rho\ x = a$ we defined:

$$\begin{aligned}\text{code}^i_{\text{R}}\ x = e\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e\ \rho\\ &\qquad \text{move}\ R_a\ R_i\end{aligned}$$

However, $\boxed{x = e;}$ is also an *expression statement*:

## About Statements and Expressions

General idea for translation:

$\text{code}^i\ s\ \rho$ : generate code for statement $s$

$\text{code}^i_{\text{R}}\ e\ \rho$ : 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:

$$\begin{aligned}\text{code}^i_{\text{R}}\ x = e\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e\ \rho\\ &\qquad \boxed{\text{move}\ R_a\ R_i}\end{aligned}$$

However, $\boxed{x = e;}$ is also an *expression statement*:

Define:

$$\boxed{\cancel{\text{code}^i}\ e_1} = e_2;\ \rho \quad = \quad \boxed{\text{code}^i_{\text{R}}}\ e_1 = e_2\ \rho$$

The temporary register $R_i$ is ignored here. More general:

$$\boxed{\text{code}^i}\ e;\ \rho = \boxed{\text{code}^i_{\text{R}}}\ e\ \rho$$

## Translation of Statement Sequences

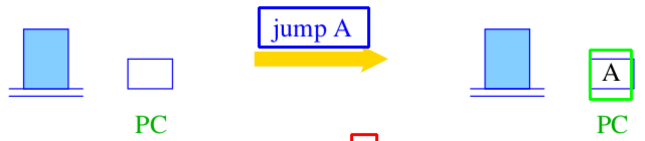
The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

$$\begin{aligned}\text{code}^i\ \boxed{(s\ ss)}\ \rho \quad &= \quad \boxed{\text{code}^i\ s\ \rho}\\ &\qquad \text{code}^i\ ss\ \rho\\ \boxed{\text{code}^i\ \varepsilon\ \rho} \quad &= \quad \boxed{\phantom{xxxx}}\ //\ \textit{empty sequence of instructions}\end{aligned}$$

Note here: $s$ is a statement, $ss$ is a sequence of statements

## Jumps

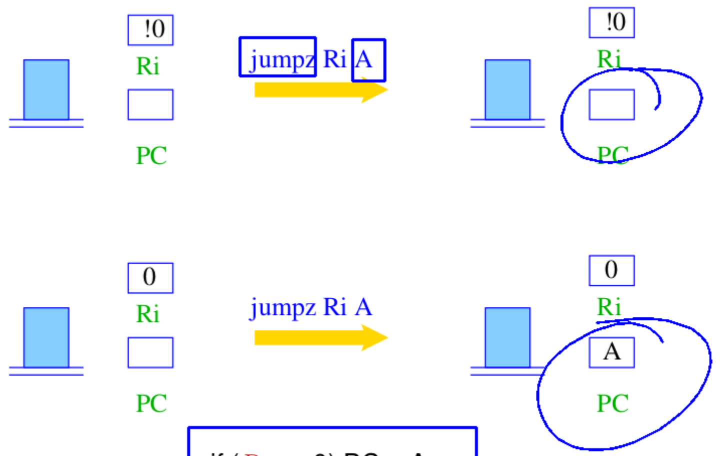
In order to diverge from the linear sequence of execution, we need *jumps*:



PC = A;

## Conditional Jumps

A conditional jump branches depending on the value in $R_i$:



if ($R_i$ == 0) PC = A;

## Simple Conditional

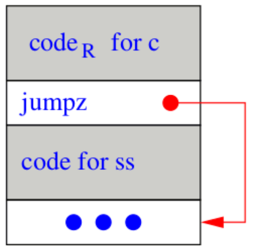
We first consider $s \equiv$ `if ( c ) ss;`
...and present a translation without basic blocks.

Idea:

emit the code of $c$ and $ss$ in sequence

insert a jump instruction in-between, so that correct control flow is ensured

$$\begin{array}{rcl} \text{code}^i\ s\ \rho & = & \text{code}^i_R\ c\ \rho \\ & & \text{jumpz}\ R_i\ A \\ & & \text{code}^i\ ss\ \rho \\ & A: & \dots \end{array}$$


code R for c
jumpz
code for ss
● ● ●