Script generated by TTT

Title: Simon: Compilerbau (23.06.2014)

Date: Mon Jun 23 14:15:15 CEST 2014

Duration: 90:10 min

Pages: 68

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

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

$\rightsquigarrow$ 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

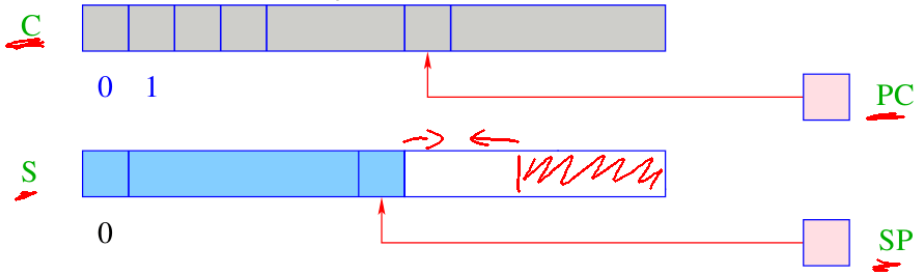
Virtual Machines

A virtual machines 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:

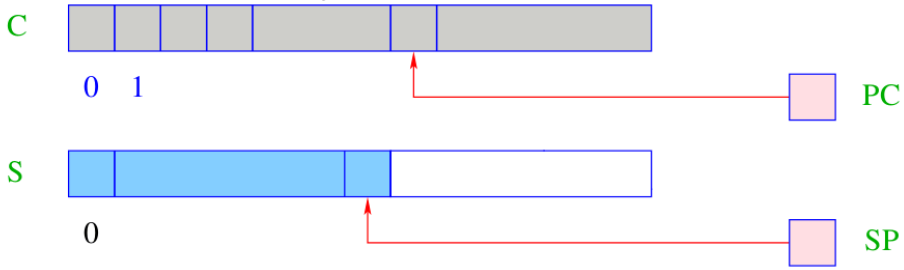


A virtual machine such as the JVM has the following structure:

- S: the data store – a memory region in which cells can be stored in LIFO order $\rightsquigarrow$ stack.
- SP: ($\widehat{=}$ 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 $\rightsquigarrow$ stack.
- SP: ($\widehat{=}$ 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 ($\widehat{=}$ program counter) address of the instruction that is to be executed next
- PC contains 0 initially

Executing a Program

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

```
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:

Evaluation of Expressions

Simple Expressions and Assignments

Task: evaluate the expression $(1 + 7) * 3$
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 $(1 + 7) * 3$
that is, generate an instruction sequence that

- computes the value of the expression and
- stores it on top of the stack

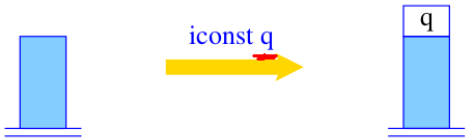
Idea:

- 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 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 op consumes its arguments
- any resulting values are stored on top of the stack

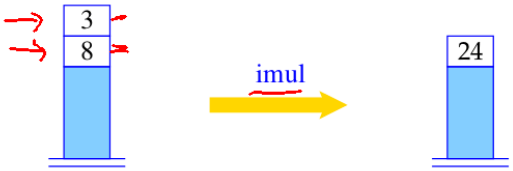


SP++;
S[SP] = q;

the instruction iconst q puts the `int`-constant q onto the stack

Binary Operators

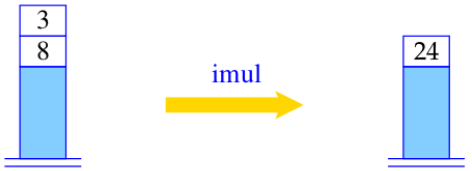
Operators with two arguments run as follows:



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

Binary Operators

Operators with two arguments run as follows:



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

- imul expects two arguments on top of the stack, consumes them and puts the result on top of the stack

Composition of Instructions

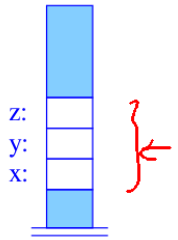
Example: generate code for $1 + 7$:

```
iconst 1    iconst 7    iadd
```

Execution of this instruction sequence:

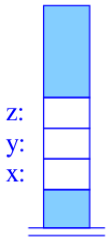
Expressions with Variables

Variables occupy a memory cell in S:



Expressions with Variables

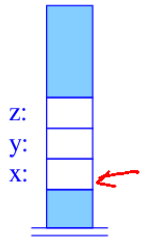
Variables occupy a memory cell in S:



- 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

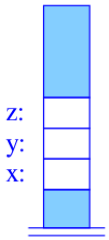
Variables occupy a memory cell in S:



- 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

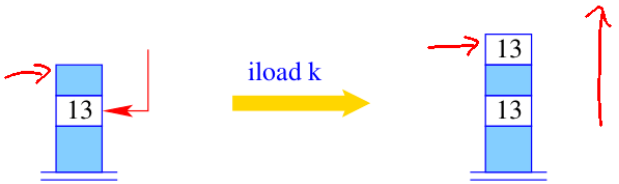
Variables occupy a memory cell in S:



- 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 ρ, that contains mappings form a variable x to the (relative) address of x; the map ρ is called *address environment* (or simply *environment*).

Reading from a Variable

The instruction iload k loads the value at address k, where k is *relative* to the top of the stack

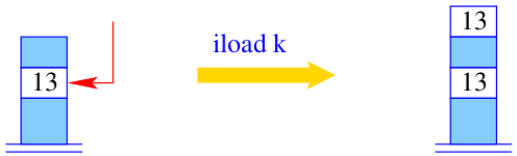


S[SP+1] = S[SP-k]; SP = SP+1;

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

Reading from a Variable

The instruction iload 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 $\rho = \{x \mapsto 1\}$:

```
iload 1
iconst 2
iadd
```

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.

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

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:

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:

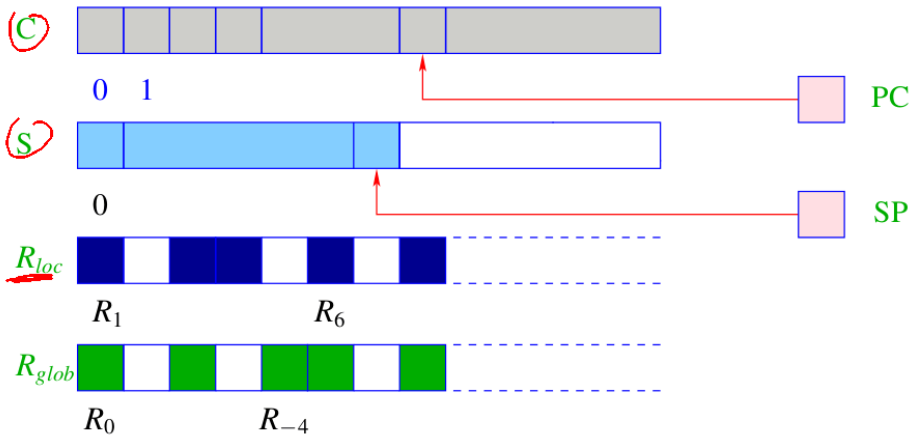
1. store variables and function arguments in registers
2. save the content of registers *onto the stack* before calling a function
3. express any arbitrary computation using *finitely* many registers

$\rightsquigarrow$ 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:

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

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

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

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:

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

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 $\rho\ x = a$):

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

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 $\rho\ x = a$):

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

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

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

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.

$$\text{op } R_i\ R_j\ R_k$$

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

Correspondingly, we generate code as follows:

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

Example: Translate $3 \star 4$ with $i = 4$:

$$\begin{aligned} \text{code}^4_{\text{R}}\ 3 \star 4\ \rho \quad = \quad & \text{code}^4_{\text{R}}\ 3\ \rho \\ & \text{code}^5_{\text{R}}\ 4\ \rho \\ & \text{mul } R_4\ R_4\ R_5 \end{aligned}$$

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.

$$\text{op } R_i\ R_j\ R_k$$

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

Correspondingly, we generate code as follows:

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

Example: Translate $3 \star 4$ with $i = 4$:

$$\begin{aligned} \text{code}^4_{\text{R}}\ 3 \star 4\ \rho \quad = \quad & \text{loadc } R_4\ 3 \\ & \text{loadc } R_5\ 4 \\ & \text{mul } R_4\ R_4\ R_5 \end{aligned}$$

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.

$$\text{op } R_i\ R_j\ R_k$$

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

Correspondingly, we generate code as follows:

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

Example: Translate $3 \star 4$ with $i = 4$:

$$\begin{aligned} \text{code}^4_{\text{R}}\ 3 \star 4\ \rho \quad = \quad & \text{code}^4_{\text{R}}\ 3\ \rho \\ & \text{code}^5_{\text{R}}\ 4\ \rho \\ & \text{mul } R_4\ R_4\ R_5 \end{aligned}$$

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:

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

Managing Temporary Registers

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

$$\begin{aligned} \text{code}^4_{\text{R}}\ \texttt{3*4+3*4}\ \rho \quad = \quad & \text{code}^4_{\text{R}}\ \texttt{3*4}\ \rho \\ & \text{code}^5_{\text{R}}\ \texttt{3*4}\ \rho \\ & \text{add}\ R_4\ R_4\ R_5 \end{aligned}$$

where

$$\begin{aligned} \text{code}^i_{\text{R}}\ \texttt{3*4}\ \rho \quad = \quad & \text{loadc}\ R_i\ 3 \\ & \text{loadc}\ R_{i+1}\ 4 \\ & \text{mul}\ R_i\ R_i\ R_{i+1} \end{aligned}$$

we obtain

$$\begin{aligned} \text{code}^4_{\text{R}}\ \texttt{3*4+3*4}\ \rho \quad = \quad & \text{loadc}\ R_4\ 3 \\ & \text{loadc}\ R_5\ 4 \\ & \text{mul}\ R_4\ R_4\ R_5 \\ & \text{loadc}\ R_5\ 3 \\ & \text{loadc}\ R_6\ 4 \\ & \text{mul}\ R_5\ R_5\ R_6 \\ & \text{add}\ R_4\ R_4\ R_5 \end{aligned}$$

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 $R_i\ R_j\ R_k$	$R_i = sgn(R_k)k$ wobei $\lvert R_j\rvert = n\lvert R_k\rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k\rvert$
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

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 $R_i\ R_j\ R_k$	$R_i = sgn(R_k)k$ wobei $\lvert R_j\rvert = n\lvert R_k\rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k\rvert$
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

Note: all registers and memory cells contain operands in $\mathbb{Z}$

Translation of Unary Operators

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

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

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.

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:

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

Applying Translation Schema for Expressions

Suppose the following function is given:

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

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

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

Applying Translation Schema for Expressions

Suppose the following function is given:

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

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

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

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, $i = 4$.

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

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, $i = 4$.

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

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

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

Code Synthesis

Chapter 4:

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

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

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

However, $x = e$ is also a *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:

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

However, $x = e$ is also a *statement*:

- Define:

$$\text{code}^i\ e_1 = e_2\ \rho = \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho$$

The temporary register R_i is ignored here. More general:

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

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:

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

However, $x = e$ is also a *statement*:

- Define:

$$\text{code}^i\ e_1 = e_2\ \rho = \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho$$

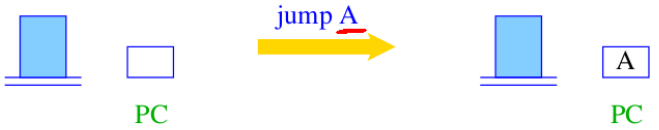
The temporary register R_i is ignored here. More general:

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

- Observation: the assignment to e_1 is a side effect of the evaluating the expression $e_1 = e_2$.

Jumps

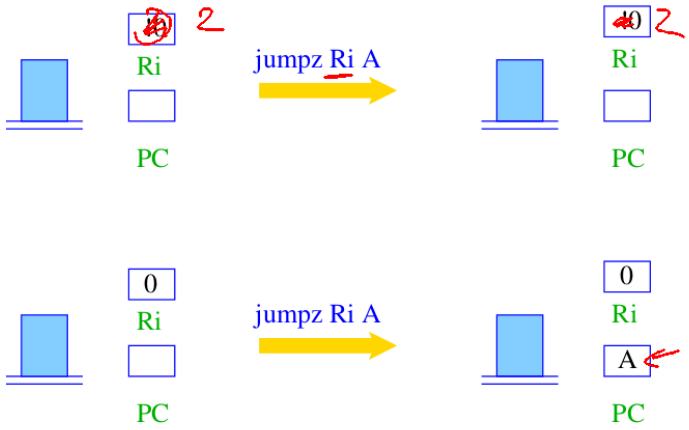
In order to diverge from the linear sequence of execution, we need *jumps*:



PC = A;

Conditional Jumps

A conditional jump branches depending on the value in R_i:



if (R_i == 0) PC = A;

Management of Control Flow

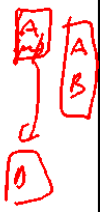
In order to translate statements with control flow, we need to emit jump instructions.

- 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)



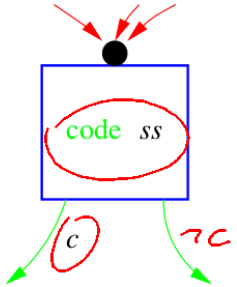
Management of Control Flow

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)
- organize instruction sequence into blocks without jumps

Basic Blocks and the Register C-Machine

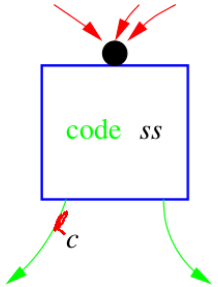
The R-CMa features only a single conditional jump, namely jumpz.



Outgoing edges must have the following form:

Basic Blocks and the Register C-Machine

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

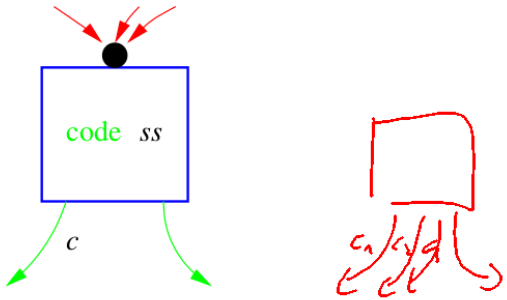


Outgoing edges must have the following form:

1. a single edge (unconditional jump), translated with jump

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 jumpi and jump (to be discussed later)

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 ($\rightsquigarrow$ 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

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 ($\rightsquigarrow$ 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

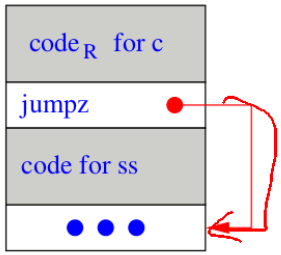
Simple Conditional

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

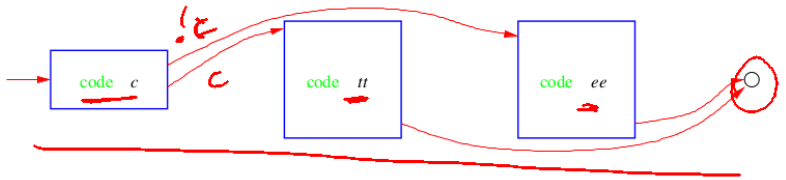
Idea:

- emit the code of c and ss in sequence
- insert a jump instruction in-between, so that correct control flow is ensured

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

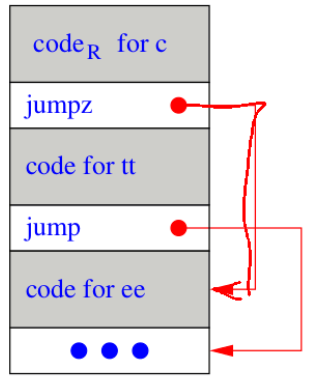


General Conditional



Translation of **if** (c) tt **else** ee.

$$
\begin{array}{lll}
\text{code}^i\ \texttt{if}(c)\ tt\ \texttt{else}\ ee\ \rho & = & \text{code}^i_R\ c\ \rho \\
 & & \text{jumpz}\ R_i\ A \\
 & & \text{code}^i\ tt\ \rho \\
 & & \text{jump}\ B \\
A: & & \text{code}^i\ ee\ \rho \\
B: & &
\end{array}
$$



Example for if-statement

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

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

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

```
code_R^i x
code_R^{i+1} y
gr R_i R_i R_{i+1}
jumpz R_i A
code^i  x = x - y
jump B
A: code^i  y = y - x
B:
```