Variables can be used in two different ways:

**Example:** \( x = y + 1 \)

We are interested in the value of \( y \), but in the address of \( x \).

The syntactic position determines, whether the L-value or the R-value of a variable is required.

<table>
<thead>
<tr>
<th>L-value of ( x )</th>
<th>address of ( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-value of ( x )</td>
<td>content of ( x )</td>
</tr>
</tbody>
</table>

**Note:**

Not every expression has an L-value (Ex.: \( x + 1 \)).
We define:

\[
\text{code}_K (e_1 + e_2) \rho = \text{code}_K e_1 \rho \\
\text{code}_K e_2 \rho \\
\text{add}
\]

... analogously for the other binary operators

\[
\text{code}_K (\neg e) \rho = \text{code}_K e \rho \\
\text{neg}
\]

... analogously for the other unary operators

\[
\text{code}_K q \rho = \text{loadc} q \\
\text{code}_K x \rho = \text{loadc} (\rho x)
\]

... The instruction `load` loads the contents of the cell, whose address is on top of the stack.

![Diagram showing load operation](image1)

\[
S[SP] = S[S[SP]];
\]

23

---

\[
\text{code}_K (x = e) \rho = \text{code}_K e \rho \\
\text{code}_K x \rho \\
\text{store}
\]

`store` writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

**Note:** this differs from the code generated by `gcc`.

![Diagram showing store operation](image2)

\[
S[S[SP]] = S[SP-1]; \\
SP--; \\
\]

25

---

\[
\text{code}_K (x = e) \rho = \text{code}_K e \rho \\
\text{code}_K x \rho \\
\text{store}
\]

`store` writes the contents of the second topmost stack cell into the cell, whose address in on top of the stack, and leaves the written value on top of the stack.

**Note:** this differs from the code generated by `gcc`.

![Diagram showing store operation](image3)

\[
S[S[SP]] = S[SP-1]; \\
SP--; \\
\]

25
Example: Code for $e \equiv x = y - 1$ with $\rho = \{e \mapsto 4, y \mapsto 7\}$.

$\text{load} \ 7 \ 	ext{load} \ 1 \ 	ext{load} \ 4$

Implements:

Introduction of special instructions for frequently used instruction sequences, e.g.,

$\text{load} \ a \ q = \text{loadc} \ q$

$\text{store} \ a \ q = \text{loadc} \ q$

3 Statements and Statement Sequences

Is $e$ an expression, then $e; \rho$ is a statement.

Statements do not deliver a value. The contents of the SP before and after the execution of the generated code must therefore be the same.

$\text{code} \ e; \rho = \text{code}_\rho \ e \rho$

$\text{pop}$

The instruction $\text{pop}$ eliminates the top element of the stack.

The code for a statement sequence is the concatenation of the code for the statements of the sequence:

$\text{code} \ (s; s; s) \rho = \text{code} \ s \rho$

$\text{code} \ s; s \rho$

$\text{code} \ e \rho = // \text{empty sequence of instructions}$
The code for a statement sequence is the concatenation of the code for the statements of the sequence:

\[
\text{code } (ss) \, \rho = \text{code } s \, \rho \\
\text{code } ss \, \rho \\
\text{code } \epsilon \, \rho = \quad \text{ // empty sequence of instructions}
\]

4 Conditional and Iterative Statements

We need jumps to deviate from the serial execution of consecutive statements:

\[
\text{PC} \quad \text{jump } A \quad \text{PC}
\]

\[
\text{PC} = A;
\]
if (6[SP] == 0) PC = A;
SP--;

For ease of comprehension, we use **symbolic jump targets**. They will later be replaced by absolute addresses.

Instead of absolute code addresses, one could generate **relative addresses**, i.e., relative to the actual PC.

**Advantages:**
- **smaller addresses** suffice most of the time;
- the code becomes **relocatable**, i.e., can be moved around in memory.
4.1 One-sided Conditional Statement

Let us first regard $s \equiv \text{if } P \text{ then } s'$.

Idea:
- Put code for the evaluation of $e$ and $s'$ consecutively in the code store,
- Insert a conditional jump (jump on zero) in between.

4.2 Two-sided Conditional Statement

Let us now regard $s \equiv (e) \ s_1 \ \text{else} \ s_2$. The same strategy yields:

$$\text{code } s \ \rho = \text{code}_R \ e \ \rho \begin{cases} \text{jumpz } A \ \\
\text{code } s_1 \ \rho \ \\
\text{jump } B \ \\
\text{code } s_2 \ \rho \ \\
\text{code } s \ 2 \end{cases} \ \\
A : \begin{align*}
\text{code } s_2 \ \rho \\
\text{jump } \rho
\end{align*} \ \\
B : \ldots$$

Example: Let $\rho = \{x \mapsto 4, y \mapsto 7\}$ and

$$s \equiv \begin{cases} \text{if } (x > y) \quad (i) \\
x = x - y; \quad (ii) \\
\text{else } y = y - x; \quad (iii) \\
\end{cases}$$

$\text{code } s \ \rho$ produces:

$$\begin{align*}
\text{loada } 4 \\
\text{loada } 7 \\
\text{gr} \\
\text{jumpz } A \\
\text{storea } 4 \\
\text{pop} \\
\text{jump } B
\end{align*} \quad \begin{align*}
A : \begin{align*}
\text{loada } 7 \\
\text{sub} \\
\text{storea } 7 \\
\text{pop} \\
\text{jump } B \\
B : \ldots
\end{align*}
$$

(i) (ii) (iii)
4.3 while-Loops

Let us regard the loop $s \equiv \text{while } (e) \ s'$. We generate:

\[
\text{code } s \ \rho = \begin{cases} \text{code}_E e \ \rho & \text{if } \text{code } \ s' \ \rho \\ \text{jumpz } B & \\ \text{code } s' \ \rho & \\ \text{jump } A & \end{cases}
\]

4.4 for-Loops

The for-loop $s \equiv \text{for } (e_1; e_2; e_3) \ s'$ is equivalent to the statement sequence $e_1; \ \text{while } (e_2) \ (s' \ e_3)$, provided that $s'$ contains no continue-statement.

We therefore translate:

\[
\text{code } s \ \rho = \begin{cases} \text{code}_E e_1 \ \rho & \text{pop} \\ \text{code}_E e_2 \ \rho & \text{jumpz } B \\ \text{code } s' \ \rho & \text{pop} \\ \text{code}_E e_3 \ \rho & \text{jump } A & \end{cases}
\]

4.5 The switch-Statement

Idea:
- Multi-target branching in constant time!
- Use a jump table, which contains at its $i$-th position the jump to the beginning of the $i$-th alternative.
- Realized by indexed jumps.
Simplification:

We only regard switch-statements of the following form:

\[
s \equiv \text{switch } (e) \{ \\
\text{case 0: } ss_0 \text{ break; } \\
\text{case 1: } ss_1 \text{ break; } \\
\vdots \\
\text{case } k - 1: ss_{k-1} \text{ break; } \\
\text{default: } ss_k \\
\}
\]

\(s\) is then translated into the instruction sequence:

40

Simplification:

We only regard switch-statements of the following form:

\[
s \equiv \text{switch } (e) \{ \\
\text{case 0: } ss_0 \text{ break; } \\
\text{case 1: } ss_1 \text{ break; } \\
\vdots \\
\text{case } k - 1: ss_{k-1} \text{ break; } \\
\text{default: } ss_k \\
\}
\]

\(s\) is then translated into the instruction sequence:

40

\[
\text{code } s \rho = \text{code } e \rho \\
\text{check } 0 \ B \\
\text{jump } D \\
\vdots \\
\text{jump } C_k \\
\text{code } ss_k \rho \\
\text{jump } D \\
\]

- The Macro \text{check } 0 \ B checks, whether the R-value of \(e\) is in the interval \([0,k]\), and executes an indexed jump into the table \(B\).
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the \text{switch}-statement.
\[ \text{code } s \rho = \text{code}_k \varepsilon \rho \]
\[ \text{check } 0 \leq k \] B
\[ \text{jump } D \]
\[ \ldots \]
\[ \text{jump } C_k \]
\[ \text{code } s_k \rho \]
\[ \text{jump } D \]

- The \textbf{Macro} \texttt{check 0 \leq k B} checks whether the R-value of \( s \) is in the interval \([0, k]\), and executes an indexed jump into the table \( B \).
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the \texttt{switch} statement.