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

<table>
<thead>
<tr>
<th><code>codeR e \rho</code></th>
<th>produces code to compute the R-value of ( e ) in the address environment ( \rho )</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>codeL e \rho</code></td>
<td>analogously for the L-value</td>
</tr>
</tbody>
</table>

**Note:**

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

\[
\text{code}_R \left( e_1 + e_2 \right) \, \rho = \text{code}_R \, e_1 \, \rho \\
\text{code}_R \, e_2 \, \rho \\
\text{add}
\]

... analogously for the other binary operators

\[
\text{code}_R \left( -e \right) \, \rho = \text{code}_R \, e \, \rho \\
\text{neg}
\]

... analogously for the other unary operators

\[
\text{code}_R \, q \, \rho = \text{loadc} \, q \\
\text{code}_L \, x \, \rho = \text{loadc} \, (\rho \, x)
\]

...
The instruction \texttt{load} loads the contents of the cell, whose address is on top of the stack.

\[
\text{code}_R x \rho = \text{code}_L x \rho
\]

\[
\text{load}
\]

\[
\text{S}[\text{SP}] = \text{S}[\text{S}[\text{SP}]];\]
\[ \text{code}_R \ (x = e) \ \rho \ = \ \text{code}_R \ e \ \rho \]
\[ \text{code}_L \ x \ \rho \]
\[ \text{store} \]

\text{store} writes the contents of the second topmost stack cell into the cell, whose address is on top of the stack, and leaves the written value on top of the stack.

\textbf{Note:} this differs from the code generated by \texttt{gcc}??

\begin{align*}
S[S[SP]] &= S[SP-1]; \\
SP--;
\end{align*}
Example: Code for $e \equiv x = y - 1$ with $\rho = \{x \mapsto 4, y \mapsto 7\}$.

code_R $e \rho$ produces:

```
loadc 7
load
loadc 1
sub
loadc 4
store
```

Improvements:

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

\[
\begin{align*}
\text{loada } q &= \text{loadc } q \\
\text{storea } q &= \text{loadc } q
\end{align*}
\]
3 Statements and Statement Sequences

Is $e$ an expression, then $e;$ 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.

$$\text{code } e; \rho = \text{code}_R \ e \ \rho$$
$$\text{pop}$$

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

```
1
SP--;
```
The code for a statement sequence is the concatenation of the code for the statements of the sequence:

\[
\text{code} (s \, ss) \, \rho \quad = \quad \text{code} \, s \, \rho \\
\quad \quad \quad \quad \quad \text{code} \, ss \, \rho \\
\text{code} \, \varepsilon \, \rho \quad = \quad \text{empty sequence of instructions}
\]
4 Conditional and Iterative Statements

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

PC \rightarrow \text{jump A} \rightarrow PC

PC = A;

PC, A
if (S[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 \( s \equiv \textbf{if} (e) s' \).

Idea:

- Put code for the evaluation of \( e \) and \( s' \) consecutively in the code store,
- Insert a conditional jump (jump on zero) in between.
\[
\text{code } s \rho = \text{code}_R e \rho \\
\text{jumpz } A \\
\text{code } s' \rho \\
A : \ldots
\]
4.2 Two-sided Conditional Statement

Let us now regard $s \equiv \textbf{if } (e) \textbf{ s}_1 \textbf{ else } s_2$. The same strategy yields:

\[
\text{code } s \rho = \text{code}_R e \rho \\
\text{jumpz } A \\
\text{code } s_1 \rho \\
\text{jump } B \\
A : \text{code } s_2 \rho \\
B : \ldots
\]
Example:  

Be \( \rho = \{ x \mapsto 4, y \mapsto 7 \} \) and 

\[
\begin{align*}
s & \equiv \begin{cases} 
\textbf{if} \ (x > y) & (i) \\
\quad x = x - y; & (ii) \\
\quad \text{else} \ y = y - x; & (iii)
\end{cases}
\end{align*}
\]

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

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

\begin{align*}
(i) & & (ii) & & (iii)
\end{align*}
4.3 while-Loops

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

$$
code s \rho =
\begin{align*}
A & : \quad \text{code}_R e \rho \\
& \quad \text{jumpz } B \\
& \quad \text{code } s' \rho \\
& \quad \text{jump } A \\
B & : \quad \ldots
\end{align*}
$$

\begin{itemize}
\item code$_R$ for e
\item jumpz
\item code for s'
\item jump
\end{itemize}
Example: Be $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ and $s$ the statement:

$$\text{while } (a > 0) \{ c = c + 1; \ a = a - b; \}$$

code $s \rho$ produces the sequence:

A: loada 7  loada 9  loada 7  B: ...
loadc 0  loadc 1  loada 8
gr  add  sub
jumpz B  storea 9  storea 7
                      pop  pop
                      jump A
4.4 for-Loops

The for-loop \( s \equiv \textbf{for} (e_1; e_2; e_3) s' \) is equivalent to the statement sequence \( e_1; \textbf{while} (e_2) \{s' e_3; \} \) – provided that \( s' \) contains no \texttt{continue}-statement. We therefore translate:

\[
\text{\texttt{code}} \ s \ \rho \ = \ \text{\texttt{code}}_R \ e_1 \\
\quad \text{pop} \\
\quad A : \ \text{\texttt{code}}_R \ e_2 \ \rho \\
\quad \quad \text{jumpz} \ B \\
\quad \text{\texttt{code}} \ s' \ \rho \\
\quad \text{\texttt{code}}_R \ e_3 \ \rho \\
\quad \text{pop} \\
\quad \text{jump} \ A
\]

B :  \ldots
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.

\[
\begin{align*}
\text{jumpi } B & \\
\text{PC} & = B + S[SP] \\
\text{SP} & \rightarrow
\end{align*}
\]
Simplification:

We only regard \texttt{switch}-statements of the following form:

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

\(s\) is then translated into the instruction sequence:
\[
\text{code } s \rho = \text{code}_R e \rho \\
\text{C}_0: \quad \text{code } ss_0 \rho \quad \text{B: jump } C_0 \\
\text{check } 0 k \text{ B} \\
\text{jump D} \\
\ldots \\
\text{jump } C_k \\
\text{C}_k: \quad \text{code } ss_k \rho \quad \text{D: } \ldots \\
\text{jump D}
\]

- The Macro \text{check } 0 k \text{ B} checks, whether the R-value of } e \text{ is in the interval } [0, k], \text{ 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 \textbf{switch}-statement.
The R-value of \( e \) is still needed for indexing after the comparison. It is therefore copied before the comparison.

This is done by the instruction \( \text{dup} \).

The R-value of \( e \) is replaced by \( k \) before the indexed jump is executed if it is less than 0 or greater than \( k \).
dup

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

SP++;
Note:

- The jump table could be placed directly after the code for the Macro check. This would save a few unconditional jumps. However, it may require to search the switch-statement twice.

- If the table starts with $u$ instead of 0, we have to decrease the R-value of $e$ by $u$ before using it as an index.

- If all potential values of $e$ are definitely in the interval $[0, k]$, the macro check is not needed.