The general principle:

- Instructions expect their arguments on top of the stack.
- Execution of an instruction consumes its operands.
- Results, if any, are stored on top of the stack.

Instruction $\text{loadq}$ needs no operand on top of the stack, pushes the constant $q$ onto the stack.

Note: the content of register $SP$ is only implicitly represented, namely through the height of the stack.

Example:

The operator $\text{leq}$

Remark: 0 represents false, all other integers true.

Unary operators $\text{neg}$ and $\text{not}$ consume one operand and produce one result.

$\text{mul}$ expects two operands on top of the stack, consumes both, and pushes their product onto the stack.

... the other binary arithmetic and logical instructions, $\text{add, sub, div, mod, and, or}$ and $\text{xor}$, work analogously, as do the comparison instructions $\text{eq, neq, le, leq, gr}$ and $\text{geq}$.
Example: Code for $1 + 7$:

```
load: 1  load: 7  add
```

Execution of this code sequence:

```
  1  7  8
```

Example: Code for $(1 + 7) \times 3$

```
load: 1  load: 7  add
```

tab <mult>

Execution of this code sequence:

```
  1  7  8
```

Variables can be used in two different ways:

Example: $x = y + 1$

We are interested in the \textit{value} of $y$, but in the \textit{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>

\texttt{code}_{\ell} \rho$ produces code to compute the R-value of $\ell$ in the address environment $\rho$.

\texttt{code}_{\ell} \rho$ analogously for the L-value.

Note:

Not every expression has an L-value (Ex.: $x + 1$).
Variables are associated with cells in $S$:

- $x$
- $y$
- $z$

Code generation will be described by some Translation Functions, code, code$_l$, and code$_r$.

Arguments: A program construct and a function $\rho$, $\rho$ delivers for each variable $x$ the relative address of $x$, $\rho$ is called Address Environment.

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>Code generation</th>
<th>Produces code to compute the R-value of $e$ in the address environment $\rho$</th>
</tr>
</thead>
<tbody>
<tr>
<td>code$_r$ $e$ $\rho$</td>
<td>analogous for the L-value</td>
</tr>
</tbody>
</table>

**Note:**

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

We define:

\[
\begin{align*}
\text{code$_r$} \ (e_1 + e_2) \ \rho &= \ \text{code$_r$} \ e_1 \ \rho \\
&\quad \text{code$_r$} \ e_2 \ \rho \\
&\quad \text{add} \\
&\quad \ldots \ \text{analogously for the other binary operators} \\
\text{code$_r$} \ (-e) \ \rho &= \ \text{code$_r$} \ e \ \rho \\
&\quad \text{neg} \\
&\quad \ldots \ \text{analogously for the other unary operators} \\
\text{code$_r$} \ q \ \rho &= \ \text{loadc} \ q \\
\text{code$_l$} \ x \ \rho &= \ \text{loadc} \ (\rho \ x)
\end{align*}
\]

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

|$S[\text{SP}] = S[\text{SP}]$;
\[
\text{code}_x \cdot \rho = \text{code}_y \cdot \rho
\]

\[
\text{load}
\]

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

\[
[S] = [S[S]_x];
\]

\[
\text{code}_x (x = c) \cdot \rho = \text{code}_x \epsilon \cdot \rho
\]

\[
\text{code}_x \cdot \rho
\]

\[
\text{store}
\]

\[
\text{store}
\]

\[
[S] = [S[S]_x];
\]

\[
\text{SP} \leftarrow 1
\]

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

\[
\begin{array}{c}
\text{loadc 7} \\
\text{loadc 1} \\
\text{loadc 4}
\end{array}
\]

\[
\begin{array}{c}
\text{load} \\
\text{add} \\
\text{store}
\end{array}
\]

Improvements:

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

\[
\begin{array}{c}
\text{loada} q = \text{loadc} q \\
\text{storea} q \leftarrow \text{loadc} q
\end{array}
\]

3 Statements and Statement Sequences

If \( e \) is 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.

\[
\text{code } e ; \cdot \rho = \text{code}_e \cdot \rho
\]

\[
\begin{array}{c}
\text{pop}
\end{array}
\]

The instruction \text{pop} eliminates the top element of the stack.
3 Statements and Statement Sequences

If $e$ is an expression, then $e;^r$ 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.

The instruction `pop` eliminates the top element of the stack.

4 Conditional and Iterative Statements

We need jumps to deviate from the serial execution of consecutive statements:
4 Conditional and Iterative Statements

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

```
P.C = A;
```

```
if (S[SP] == 0) PC = A;
SP--;
```
4.1 One-sided Conditional Statement

Let us first regard $s = \text{if } (e) s'$.

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

\[
\text{code } s \ ho = \begin{cases} \text{code}_e \ ho \\
\text{jumpz } A \\
\text{code } s' \ ho \\
A : \ldots
\end{cases}
\]
### 4.2 Two-sided Conditional Statement

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

\[
\text{code } s \rho = \begin{cases} 
\text{code}_e e \rho \\
\text{jump } A \\
\text{code } s_1 \rho \\
\text{jump } B \\
A : \text{code } s_2 \rho \\
B : \ldots 
\end{cases}
\]

**Example:**

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

\[
s = \begin{cases} 
\text{if}(x > y) \\
x = x - y; \\
\text{else } y = y - x; 
\end{cases}
\]

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

\[
\begin{align*}
(\text{i}) & : \text{loada } 4 \\
(\text{ii}) & : \text{loada } 7 \\
(\text{iii}) & : \text{loada } 7
\end{align*}
\]

### 4.3 while-Loops

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

\[
\text{code } s \rho = \begin{cases} 
\text{code}_e e \rho \\
\text{jump } A \\
\text{code } s' \rho \\
\text{jump } A \\
A : \text{code } s' \rho \\
B : \ldots 
\end{cases}
\]

**Example:**

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

\[
s = \begin{cases} 
\text{if}(x > y) \\
x = x - y; \\
\text{else } y = y - x; 
\end{cases}
\]

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

\[
\begin{align*}
(\text{i}) & : \text{loada } 4 \\
(\text{ii}) & : \text{loada } 7 \\
(\text{iii}) & : \text{loada } 7
\end{align*}
\]
Example: \( \rho = \{ x \mapsto 4, y \mapsto 7 \} \) and
\[
\begin{align*}
    s &= \text{if } (x > y) & (i) \\
        & \quad x = x - y; & (ii) \\
        & \quad \text{else } y = y - x; & (iii)
\end{align*}
\]

code \( s \rho \) produces:

\[
\begin{align*}
\text{loada} & \quad \text{loada} 4 & A: \quad \text{loada} 7 \\
\text{loada} 7 & \quad \text{loada} 7 & \text{loada} 4 \\
\text{gr} & \quad \text{sub} & \text{sub} \\
\text{jumpe} & \quad \text{storea} 4 & \text{storea} 7 \\
\text{pop} & \quad \text{pop} & \text{pop} \\
\text{jump} & \quad \text{B:} & \ldots
\end{align*}
\]

(i) \quad (ii) \quad (iii)

4.3 while-Loops

Let us regard the loop \( s = \text{while } (c) \ s' \). We generate:

\[
\begin{align*}
\text{code } s \rho &= \quad \text{code } e \rho \\
A: \quad \text{code } e \rho \\
& \quad \text{jumpe } \\
& \quad \text{code } s' \rho \\
B: \quad \text{jump } A \\
& \quad \text{jump } B \quad \ldots
\end{align*}
\]

4.4 for-Loops

The for-loop \( s = \text{for } (c_1; c_2; c_3) \ s' \) is equivalent to the statement sequence \( c_1 \) while \( (c_2) \{s' \ c_3\} \) — provided that \( s' \) contains no continue-statement.

We therefore translate:

\[
\begin{align*}
\text{code } s \rho &= \quad \text{code } c_1 \\
A: \quad \text{code } e_2 \rho \\
& \quad \text{jumpe } \\
& \quad \text{code } s' \rho \\
& \quad \text{code } e_3 \rho \\
B: \quad \text{jump } A \\
& \quad \text{jump } B \quad \ldots
\end{align*}
\]
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.

```plaintext
PC \rightarrow \text{jumpi B} \rightarrow \text{i + SP[SP]} \rightarrow \text{PC}
```

$\text{PC} \leftarrow \text{i + SP[SP]}$

SP--;