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

The `mul` instruction expects two operands on top of the stack, consumes both, and pushes their product onto the stack.

... the other binary arithmetic and logical instructions, `add`, `sub`, `div`, `mod`, `and`, `or` and `xor`, work analogously, as do the comparison instructions `eq`, `neq`, `le`, `leq`, `gr` and `geq`. 
Example: The operator \( \text{leq} \)

\[
\begin{array}{c}
\begin{array}{c}
7 \\
3
\end{array} \quad \text{leq} \quad \begin{array}{c}
1
\end{array}
\end{array}
\]

Remark: 0 represents \textit{false}, all other integers \textit{true}.

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

\[
\begin{array}{c}
\begin{array}{c}
8
\end{array} \quad \text{neg} \quad \begin{array}{c}
-8
\end{array}
\end{array}
\]

\[S[SP] = -S[SP];\]
Example: Code for $1 + 7$:

\[\text{loadc 1} \quad \text{loadc 7} \quad \text{add}\]

Execution of this code sequence:
Variables are associated with memory cells in $S$:

$\rho$ 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.

\[
\begin{align*}
\text{L-value of } x &= \text{address of } x \\
\text{R-value of } x &= \text{content of } x
\end{align*}
\]

| \( \text{code}_R \ e \ \rho \) | produces code to compute the R-value of \( e \) in the address environment \( \rho \) |
| \( \text{code}_L \ e \ \rho \) | analogously for the L-value |

**Note:**

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

\[
\begin{align*}
\text{code}_R (e_1 + e_2) \rho & = \text{code}_R e_1 \rho \\
& \quad \text{code}_R e_2 \rho \\
& \quad \text{add} \\
& \quad \ldots \text{analogously for the other binary operators}
\end{align*}
\]

\[
\begin{align*}
\text{code}_R (-e) \rho & = \text{code}_R e \rho \\
& \quad \text{neg} \\
& \quad \ldots \text{analogously for the other unary operators}
\end{align*}
\]

\[
\begin{align*}
\text{code}_R q \rho & = \text{loadc} q \\
\text{code}_L x \rho & = \text{loadc} (\rho x) \\
& \quad \ldots
\end{align*}
\]
The instruction \texttt{load} loads the contents of the cell, whose address is on top of the stack.

\begin{align*}
\text{code}_R \ x \ \rho &= \text{code}_L \ x \ \rho
\end{align*}

\texttt{load}

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

**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.

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

\[
S[S[SP]] = S[SP-1]; \\
SP--; \\
\]
Example: Code for \( e \equiv x = y - 1 \) with \( \rho = \{ x \mapsto 4, y \mapsto 7 \} \).
\( \text{code}_R \ e \ \rho \) produces:

\[
\begin{array}{ccc}
\text{loadc 7} & \text{loadc 1} & \text{loadc 4} \\
\text{load} & \text{sub} & \text{store}
\end{array}
\]

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 \( \text{SP} \) before and after the execution of the generated code must therefore be the same.

\[
\text{code } e; \, \rho \quad = \quad \text{code}_R \, e \, \rho \\
\quad \quad \quad \text{pop}
\]

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

\[
\begin{array}{c}
\text{1} \\
\text{-------------------} \\
\text{SP--;}
\end{array}
\]
The code for a statement sequence is the concatenation of the code for the statements of the sequence:

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

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

\[ PC = A; \]
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 \( s \equiv \text{if} \ (e) \ s' \).

Idea:

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

\[
\text{code } s \ \rho \quad = \quad \text{code}_R \ e \ \rho \\
\text{jumpz} \ A \\
\text{code } s_1 \ \rho \\
\text{jump} \ B \\
A : \quad \text{code } s_2 \ \rho \\
B : \quad \ldots
\]
Example: \[ \rho = \{ x \mapsto 4, y \mapsto 7 \} \] and

\[ s \equiv \text{if} (x > y) \quad (i) \]
\[ x = x - y; \quad (ii) \]
\[ \text{else} \quad y = y - x; \quad (iii) \]

code \( s \rho \) produces:

\[
\begin{array}{llll}
\text{loada 4} & \text{loada 4} & \text{A: loada 7} \\
\text{loada 7} & \text{loada 7} & \text{loada 4} \\
\text{gr} & \text{sub} & \text{sub} \\
\text{jumpz A} & \text{storea 4} & \text{storea 7} \\
\text{pop} & \text{pop} \\
\text{jump B} & \text{B: \ldots} \\
\end{array}
\]
4.3 while-Loops

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

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

\[
\begin{align*}
A: & \quad \text{loada 7} \quad \text{loada 9} \quad \text{loada 7} \\
& \quad \text{loadc 0} \quad \text{loadc 1} \quad \text{loada 8} \\
& \quad \text{gr} \quad \text{add} \quad \text{sub} \\
& \quad \text{jumpz B} \quad \text{storea 9} \quad \text{storea 7} \\
& \quad \text{pop} \quad \text{pop} \quad \text{jump A}
\end{align*}
\]
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:

\[
\begin{align*}
\text{code } s \ \rho & = \ \text{code}_R \ e_1 \ \rho \\
& \quad \text{pop} \\
A : \quad & \text{code}_R \ e_2 \ \rho \\
& \quad \text{jumpz B} \\
& \quad \text{code} \ s' \ \rho \\
& \quad \text{code}_R \ e_3 \ \rho \\
& \quad \text{pop} \\
& \quad \text{jump } A \\
B : \quad & \ldots
\end{align*}
\]
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.

PC = B + S[SP];
SP--;