**Section 1. Concrete Semantics**

We define a toy programming language, called WHILE, which has the following syntax:

$$ S \to x := E \mid S ; S \mid \text{skip} $$

$$ \mid \text{if } E \text{ then } S \text{ else } S $$

$$ \mid \text{while } E \text{ do } S $$

$$ E \to x \mid \# \mid E \diamond E $$

A basic statement in WHILE is either an assignment, a sequential composition of two statements (separated by semicolon), or a “skip” (no-op). Control structures are branching conditional (if-then-else) and loops (while). An expression is either a single variable name (x), a number literal (#), or it can be composed from two sub-expressions via a binary operator $\diamond$. We allow the following operators:

$$ \diamond \in \{+, -, *, /, =, \neq, <, >, \leq, \geq\} $$

For simplicity of the presentation, assume that all expressions have type $\mathbb{Z}$, that is, they are signed integers. Comparison operators are supposed to return either false=0 or true=1.

A **concrete state** of a program is defined as a mapping from variables to their concrete integer values, written:

$$ \sigma : \text{Var} \to \mathbb{Z} $$

Where Var is the set of all program variables. The set containing all stores of this form is denoted by $\Sigma$.

To define the semantics of statements in WHILE, we introduce the notation

$$ [s] : \Sigma \to \Sigma $$

That is, it maps a concrete state $\sigma$ to a new concrete state $[s]\sigma$, that is reached after executing the statement $s$ on the state $\sigma$.

We now list the definition of $[\ ]$ by enumerating all the possible cases for a statement $s$.

<table>
<thead>
<tr>
<th>Case</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[x := e]\sigma$</td>
<td>$\sigma[x \mapsto [e]\sigma]$ assignment to a single variable</td>
</tr>
<tr>
<td>$[s_1 ; s_2]\sigma$</td>
<td>$[s_2][[s_1]\sigma]$ sequential composition</td>
</tr>
<tr>
<td>$[\text{skip}]\sigma$</td>
<td>$\sigma$ no-op</td>
</tr>
<tr>
<td>$[\text{if } e \text{ then } s_1 \text{ else } s_2]\sigma$</td>
<td>$[s_1]\sigma$ if $[e]\sigma = \text{true}$, $[s_2]\sigma$ if $[e]\sigma = \text{false}$. conditional statements</td>
</tr>
</tbody>
</table>

Where $[e] : \Sigma \to \mathbb{Z}$ is the semantic value of expressions, and is also defined by case analysis:

<table>
<thead>
<tr>
<th>Case</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[x]\sigma$</td>
<td>$\sigma(x)$ for a single variable $x$</td>
</tr>
<tr>
<td>$[c]\sigma$</td>
<td>$c$ for a numerical constant $c$</td>
</tr>
<tr>
<td>$[e_1 \diamond e_2]\sigma$</td>
<td>$[e_1]\sigma \diamond [e_1]\sigma$ for a binary operation $\diamond$</td>
</tr>
</tbody>
</table>
Notice that we have forgone loop statements. The definition for loops is a bit more convoluted:
When \( \llbracket e \rrbracket \sigma = \text{false} \), \( \llbracket \text{while } e \text{ do } s \rrbracket \sigma = \sigma \) (a statement that has no effect).
When \( \llbracket e \rrbracket \sigma = \text{true} \), running \( \text{while } e \text{ do } s \) has the same effect as \( s ; \text{while } e \text{ do } s \).
Therefore,
\[
\llbracket \text{while } e \text{ do } s \rrbracket \sigma = [s ; \text{while } e \text{ do } s ](\llbracket s \rrbracket \sigma)
\]
It can be seen that the semantic definition of “while” is given in terms of itself.
Let \( w = [\text{while } e \text{ do } s] \), we can express it as the equality:
\[
w(\sigma) = \begin{cases} 
\sigma & \text{if } \llbracket e \rrbracket \sigma = \text{false}, \\
w([s] \sigma) & \text{if } \llbracket e \rrbracket \sigma = \text{true}. 
\end{cases}
\] (1)

For loops that always terminate, there can be only one possible solution for \( w \). (This can be shown by induction on the number of iterations the loop takes.) However, for nonterminating executions, may be multiple or even infinitely many solutions.

As an example, suppose \( e = (n \neq 0) \) and \( s = (n := n - 1) \), that is, the statement is \( \text{while } n \neq 0 \text{ do } n := n - 1 \)
Clearly, for any nonnegative initial value of \( n \), the loop terminates with \( n = 0 \). However, for negative values, the loop diverges, \( i.e. \) never terminates. We denote nontermination with \( \perp \), meaning that the semantic function is undefined for these inputs.
\[
w(\sigma) = \begin{cases} 
\sigma[n \mapsto 0] & \text{if } \sigma(n) \geq 0, \\
\perp & \text{if } \sigma(n) < 0.
\end{cases}
\]
Notice that this \( w \) satisfies equation (1). However, here is another definition that also satisfies it:
\[
w(\sigma) = \begin{cases} 
\sigma[n \mapsto 0] & \text{if } \sigma(n) \geq 0, \\
\sigma[n \mapsto 42] & \text{if } \sigma(n) < 0.
\end{cases}
\]

The ambiguity stems from the fact that for states satisfying \( \sigma(n) < 0 \), it is also true that \( ([s] \sigma)(n) < 0 \).
and in both cases, \( \llbracket e \rrbracket \sigma = [e][[s] \sigma] = \text{true} \). Equation (1) alone is not sufficient to enforce the value to be \( \perp \); all it requires is of the values of \( w \) to be the same on the two states — which clearly holds for both \( \perp \) and \( \sigma[n \mapsto 42] \).

The solution is to define the “true” semantics of while-loops as the least solution that satisfies (1). In this respect, \( \perp \subseteq \sigma \) for any state \( \sigma \), and we say therefore that \( w \subseteq w^o \). This kind of semantics is called least fixed point semantics.
Section 2. Galois Connection

As a prelude to defining semantics for abstract interpretation, we define the notion of a relationship between lattices known as Galois connection.

Definition. Let $C$, $A$ be two lattices and let $\alpha : C \to A$ and $\gamma : A \to C$ be monotone functions between them. Then, $\alpha$, $\gamma$ form a Galois connection when the following holds:

$$\forall c \in C, a \in A. \quad \alpha(c) \sqsubseteq a \iff c \sqsubseteq \gamma(a) \quad (2)$$

Or, equivalently (since (2) is somewhat hard to grasp intuitively),

$$\forall c \in C, a \in A. \quad \alpha(\gamma(a)) \sqsubseteq a \land c \sqsubseteq \gamma(\alpha(c)) \quad (3)$$

Which states, that applying $\gamma$ and then $\alpha$, can only push the argument lower (in $A$), whereas applying $\alpha$ followed by $\gamma$ can only lift it higher (in $C$).

It can be shown easily that (2) implies (3):

$$\gamma(a) \sqsubseteq \gamma(a) \Rightarrow \alpha(\gamma(a)) \sqsubseteq a \quad \text{(from reflexivity of } \sqsubseteq \text{ and from (2), going right to left, with } c = \gamma(a))$$

$$\alpha(c) \sqsubseteq \alpha(c) \Rightarrow c \sqsubseteq \gamma(\alpha(c)) \quad \text{(from reflexivity of } \sqsubseteq \text{ and from (2), going left to right, with } a = \alpha(c))$$

The converse, that (3) implies (2), can also be derived based on the monotonicity of $\alpha$ and $\gamma$:

$$\alpha(c) \sqsubseteq a \Rightarrow c \sqsubseteq \gamma(\alpha(c)) \sqsubseteq \gamma(a) \Rightarrow c \sqsubseteq \gamma(a) \quad \text{(recall that } \sqsubseteq \text{ is transitive)}$$

$$c \sqsubseteq \gamma(a) \Rightarrow \alpha(c) \sqsubseteq \alpha(\gamma(a)) \sqsubseteq a \Rightarrow \alpha(c) \sqsubseteq a$$

In our context, we assume from now on that $C$ is the power-set lattice $\langle \mathcal{P} \Sigma, \subseteq \rangle$, that is, elements of $C$ are sets of concrete states, and they are ordered by set inclusion. We refer to $C$ as the concrete domain, and to $A$ as the abstract domain. Furthermore, $\alpha$ is our abstraction function, and $\gamma$ is our concretization function.

As a running example, consider a lattice $A$ comprising of the elements $\{\bot, 0, +, -, \top\}$ ordered according to the following Hasse diagram:

We can define a Galois connection as follows:

**Concretization**

$$\gamma(\bot) = \emptyset, \gamma(\top) = \mathbb{Z}, \gamma(0) = \{0\}$$

$$\gamma(+) = \{0, 1, 2, \ldots\} = \{v \in \mathbb{Z} \mid v \geq 0\} \quad (4)$$

$$\gamma(-) = \{0, -1, -2, \ldots\} = \{v \in \mathbb{Z} \mid v \leq 0\}$$

Notice that in our concretization, $0$ is included in both $\gamma(\top)$ and $\gamma(\bot)$. This is in line with the ordering of the abstract domain: $0 \subseteq (+)$ and also $0 \subseteq (-)$.

**Abstraction**

We define an auxiliary function $\beta : \mathbb{Z} \to A$ that “abstract” single values.

$$\beta(v) = \begin{cases} 
0 & v = 0 \\
+ & v > 0 \\
- & v < 0 
\end{cases}$$

Then the abstraction function is defined via: $\alpha(S) = \cup \{\beta(\sigma) \mid \sigma \in S\}$
Note: the abstraction function can be defined in this manner for any abstract domain. Moreover, once $\beta$ has been defined, the concretization function is determined by

$$
\gamma(a) = \{ \sigma \in \Sigma \mid \beta(\sigma) \sqsubseteq a \}
$$

In the above case, where $\Sigma = \mathbb{Z}$, this definition coincides with $\gamma$ as defined by (4).

We can show that $\alpha$, $\gamma$ that we just defined for $C = \mathcal{P}(\mathbb{Z})$ and $A = \{\bot, 0, +, -, \top\}$ indeed form a Galois connection:

E.g. for $a = (+)$,

$$
\alpha(c) \sqsubseteq (+) \iff \sqcup \{ \beta(\sigma) \mid \sigma \in c \sqsubseteq (+) \} \iff \forall \sigma \in c, \beta(\sigma) \sqsubseteq (+) \iff \forall \sigma \in c, \sigma \geq 0 \iff c \sqsubseteq \{0,1,2,...\}
$$

The steps are then repeated for $a = (-)$, $a = (0)$.

For $a = (\bot)$, and $a = (\top)$, the proof is always trivial: $\alpha(c) \sqsubseteq \top$ and $c \sqsubseteq \gamma(\top)$ are both true statements unconditionally, and $\alpha(c) \sqsubseteq \bot$ and $c \sqsubseteq \gamma(\bot)$ are both true iff $c = \emptyset$. 
Section 3. Abstract Semantics

Now that we have concrete semantics $[s]$ and a connection between concrete sets of stores $C$ and abstract states $A$, it is time to define the abstract interpreter.

We go back to our statements in the language WHILE, and introduce the notation:

$$[s] : A \rightarrow A$$

The symbol # (pronounced: “sharp”) signifies that this new semantics operates on abstract states. The expression $[s]#\sigma$ signifies the abstract state obtained from executing the statement $s$ on an initial, also abstract, state $\sigma$.

Intuitively, an abstract state $\sigma$ will serve as a compact representation of a set of concrete states given by $\gamma(\sigma)$. For example, in a WHILE program with $k$ variables $v_1, v_2, \ldots, v_k$, concrete stores can be represented as $\sigma \in \mathbb{Z}^k$, and abstract states can be represented as e.g.,

$$\sigma^# \in A = \{\bot, 0, +, -, \top\}^k$$

Of course, the choice of abstract domain depends greatly on the properties that we wish to prove of our programs. Using the domain above, where each variable is associated with an element of $\{\bot, 0, +, -, \top\}$, would be suitable for proving properties about the sign (positive, negative) of the values stored in program variables.

As in the case of concrete semantics, the abstract semantics is defined by case analysis on the structure of the statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Abstract Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := e$</td>
<td>$[x := e]#\sigma = \sigma[x \mapsto [e]#\sigma]$ assignment to a single variable</td>
</tr>
<tr>
<td>$[s_1; s_2]$</td>
<td>$[s_1; s_2]# = [s_2]#([s_1]#\sigma)$ sequential composition</td>
</tr>
<tr>
<td>$[\text{skip}]$</td>
<td>$[\text{skip}]#\sigma = \sigma#$ no-op</td>
</tr>
</tbody>
</table>

Where we also introduced a new notation $[e] : A \rightarrow \{\bot, 0, +, -, \top\}$ for the abstract semantics of expressions:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Abstract Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[x]#\sigma = \gamma(x)$</td>
<td>for a single variable $x$</td>
</tr>
<tr>
<td>$[c]#\sigma = \beta(c)$</td>
<td>for a numerical constant $c$</td>
</tr>
<tr>
<td>$[e_1 \diamond e_2]#\sigma = [e_1]#\sigma \diamond [e_2]#\sigma$</td>
<td>for a binary operation $\diamond$</td>
</tr>
</tbody>
</table>

Notice the “hat” on top of the operator $\diamond$, telling us that this is not the “normal” arithmetic operation operating on numbers, but rather an abstract operation whose operands are from the abstract value domain, $\{\bot, 0, +, -, \top\}$.
As an example, consider integer multiplication, ‘\(\cdot\)’. The abstract variant \(\sqcap\) is given by the following “multiplication” table:

<table>
<thead>
<tr>
<th>(\sqcap)</th>
<th>(\bot)</th>
<th>0</th>
<th>+</th>
<th>−</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bot)</td>
<td>(\bot)</td>
<td>(\bot)</td>
<td>(\bot)</td>
<td>(\bot)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>0</td>
<td>(\bot)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>(\bot)</td>
<td>0</td>
<td>+</td>
<td>−</td>
<td>T</td>
</tr>
<tr>
<td>−</td>
<td>(\bot)</td>
<td>0</td>
<td>−</td>
<td>+</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>(\bot)</td>
<td>0</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

The row and column pertaining to \(\bot\) are somewhat immaterial, as they will always contain \(\bot\) as a result (if one of the operands of an expression is undefined, than the result is undefined). Of the other entries, 0 multiplied by whatever value (even \(\top\)) gives 0, and the sign of the product can be deduced by the signs of the factors if they are both known (equal sign = positive, unequal signs = negative).

Similar tables can be constructed for other operators as well. In some cases, the result will have to be (conservatively) coerced to \(\top\); for example, \((+) \sqcap (+) = (+)\), but \((+) \sqcap (−) = T\).