Programming Languages and Computation

Week 7–8: Executing the While Language

In this worksheet, you will build upon your parser for the While programming language by implementing the denotational and operational semantics. By the end of this sheet, you should have completely implemented your own programming language! As with the parser, you may use any programming language of your choice, however the instructions are most easily followed in a non-pure functional programming language such as OCaml, or an imperative/object-oriented programming language.

Task 1: Denotational Semantics

The first task is to implement the denotational semantics for arithmetic and Boolean expressions. Recall, that the denotational semantics for arithmetic expressions is given as a function $[\![\cdot]\!]_{\mathcal{A}} : \mathcal{A} \to (\mathsf{State} \to \mathbb{Z})$ that maps arithmetic expressions to functions from states to the integers. It is defined by recursion over the structure of expressions as follows:

$$\begin{aligned}
[\![x]\!]_{\mathcal{A}}(\sigma) &= \sigma(x) \\
[\![n]\!]_{\mathcal{A}}(\sigma) &= n \\
[\![e_1 + e_2]\!]_{\mathcal{A}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) + [\![e_1]\!]_{\mathcal{A}}(\sigma) \\
[\![e_1 - e_2]\!]_{\mathcal{A}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) - [\![e_2]\!]_{\mathcal{A}}(\sigma) \\
[\![e_1 * e_2]\!]_{\mathcal{A}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) \times [\![e_2]\!]_{\mathcal{A}}(\sigma)
\end{aligned}$$

In this definition, the parameter $\sigma \in \mathsf{State}$ is the state of the program under which the expression is being evaluated. Recall that our mathematical representation for states is as functions from variables to integers: $\mathsf{State} = \mathsf{Var} \to \mathbb{Z}$.

1. Define a type for program states that represents functions from variables to integers. If you're doing this in OCaml, you may wish to use a *type synonym* using the syntax:

   ```
   type my_type_synonym = the_type
   ```

   If you want to implement a more realistic interpreter, consider using a *hash-table* instead of a function - https://ocaml.org/docs/hash-tables. Hash tables can be thought of as an efficient representation of a partial function with a finite domain.

2. Write a function *lookup_var* that takes a state and a variable and returns the integer assigned to that variable in the given state. If you're using a hash table, consider what value should be associated to variables not present in the table.

3. Next, we will need to define our denotation function for arithmetic expressions *evaluate_arithmetic*. This function should take an expression $e \in \mathcal{A}$ and a state $\sigma \in \mathcal{S}\llcorner\dashv\lrcorner\rceil$ as inputs and return the integer value of that expression when evaluated in the given according to the denotation function, i.e. $[\![e]\!]_{\mathcal{A}}(\sigma)$.

If you are implementing the interpreter in a function language (e.g. OCaml or Haskell) this function should be defined *recursion* over the syntax tree, mimicking the definition of the denotation function itself. For an object-oriented approach, it may make sense to add an additional method to the class of expressions, or consider using the visitor pattern.

Keep in mind that our syntax trees were defined using a single type for both arithmetic and Boolean expressions in the previous sheet. Therefore, this function will likely be a partial function that throws an error when given a Boolean expression.

Recall that denotational semantics for Boolean expressions is defined similarly as a function $[\![\cdot]\!]_{\mathcal{B}} : \mathcal{B} \to (\mathsf{State} \to \mathbb{Z})$ that recurses over the input expression:

$$\begin{aligned}
[\![\mathsf{true}]\!]_{\mathcal{B}}(\sigma) &= \top \\
[\![\mathsf{false}]\!]_{\mathcal{B}}(\sigma) &= \bot \\
[\![e_1 \leq e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) \leq [\![e_2]\!]_{\mathcal{A}}(\sigma) \\
[\![e_1 = e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) = [\![e_2]\!]_{\mathcal{A}}(\sigma) \\
[\![!e]\!]_{\mathcal{B}}(\sigma) &= \neg [\![e]\!]_{\mathcal{B}}(\sigma) \\
[\![e_1 \;\&\&\; e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{B}}(\sigma) \wedge [\![e_1]\!]_{\mathcal{B}}(\sigma) \\
[\![e_1 \;||\; e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{B}}(\sigma) \;||\; [\![e_2]\!]_{\mathcal{B}}(\sigma)
\end{aligned}$$

4. Repeat the same process for Boolean expressions, defining a function *evaluate_boolean* that take an expression and a state as inputs and return a Boolean value computed by recursion over the syntax tree. The function should use the *evaluate_arithmetic* function on arithmetic sub-expressions in order to evaluate them.

Task 2: Operational Semantics

In this task you will implement the operational semantics for While statements. The operational semantics is formally given as a relation $\Downarrow \subseteq \mathcal{S} \times \mathsf{State} \times \mathsf{State}$ that is defined inductively by the following inference rules:

$$\frac{}{\mathsf{skip}, \sigma \Downarrow \sigma} \qquad \frac{}{x \leftarrow e, \sigma \Downarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)]}$$

$$\frac{S_1, \sigma_1 \Downarrow \sigma_2 \quad S_2, \sigma_2 \Downarrow \sigma_3}{S_1; S_2, \sigma_1 \Downarrow \sigma_3} \qquad \frac{S_1, \sigma_1 \Downarrow \sigma_2}{\mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \sigma_1 \Downarrow \sigma_2}\, [\![e]\!]_{\mathcal{B}}(\sigma_1) = \top$$

$$\frac{S_2, \sigma_1 \Downarrow \sigma_2}{\mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \sigma_1 \Downarrow \sigma_2}\, [\![e]\!]_{\mathcal{B}}(\sigma_1) = \bot \qquad \frac{S, \sigma_1 \Downarrow \sigma_2}{\mathsf{while}\ e\ \mathsf{do}\ S, \sigma_1 \Downarrow \sigma_2}\, [\![e]\!]_{\mathcal{B}}(\sigma_1) = \bot$$

$$\frac{S, \sigma_1 \Downarrow \sigma_2 \quad \mathsf{while}\ e\ \mathsf{do}\ S, \sigma_2 \Downarrow \sigma_3}{\mathsf{while}\ e\ \mathsf{do}\ S, \sigma_1 \Downarrow \sigma_3}\, [\![e]\!]_{\mathcal{B}}(\sigma_1) = \top$$

5. Before implementing the operational semantics, we need a function for updating the state that can be used to interpret the assignment statement. This function, *update_var*, should take as input a state $\sigma \in \mathsf{State}$, a variable $x \in \mathsf{Var}$, and a new value $n \in \mathbb{Z}$ and returns the state $\sigma[x \mapsto n]$, i.e. the state that maps x to n and is the same as σ with regards to all other variables.

Now we are ready to implement the operational semantics. The relation $\Downarrow \subseteq \mathcal{S} \times \text{State} \times \text{State}$ can be used to evaluate the behaviour of a program by attempting to solve the following problem: given a statement $S \in \mathcal{S}$ and an initial state $\sigma \in \text{State}$, find a final state $\sigma' \in \text{State}$ such that $S, \sigma \Downarrow \sigma'$. The final state is computed by applying the relevant inference rules to implicitly construct a derivation.

For example, to compute the behaviour of the program if e then S_1 else S_2 we need to consider which inference rules are applicable. There are two rules that govern the behaviour of the if construct and these depend on whether or not the Boolean expression e is true in the current state. If it is true, we can compute the final state by executing the statement S_1 in the same initial state. This particular inference rule would correspond to the code fragment below:

```
let execute_statement (s : stmt) (sigma : state) : state
  match s with
  | ...
  | Cond (e, s1, s2) ->
      if evaluate_arithmetic e sigma
        then execute_statement s1 sigma
        else ...
  | ...
```

Notice that the premises of inference rules require us to simulate the behaviour of other expressions, for which we can recursively call our evaluation function.

6. Implement the function *execute_statement* that takes a statement $S \in \mathcal{S}$ and a state $\sigma \in \text{State}$ and computes a final state $\sigma' \in \text{State}$ such that $S, \sigma \Downarrow \sigma'$ by following the inference rules for the operational semantics.

Task 3: Integration & Testing

So far, you have created a parser that is capable of translating some input text into an abstract syntax tree of the While language (see Task 3 of the previous sheet). In this sheet, you have created an interpreter for While programs by implementing the denotational semantics for arithmetic and Boolean expressions and the operational semantics for statements. Now you should integrate these components.

7. Define a function *run_program* that takes as input a textual representation of a While program and uses the parser written in the last sheet to convert it into an abstract syntax tree that is then executed by the *execute_statement* function. You will need to pick an initial value for the state, e.g. the state that maps all variables to 0.

8. Write a While program that computes the factorial of a given number. That is, construct a statement $S \in \mathcal{S}$ such that $S, [x \mapsto n] \Downarrow \sigma$ where $\sigma(x) = n!$.

9. Test your interpreter on the While program you created in the previous exercise.