HW 3: Syntax and Operational Semantics for Ruse

CS 538, Spring 2020

March 5, 2020

1 Syntax for Ruse

In this section, we present the formal syntax for the Ruse programming language. We use a few notations to simplify the presentation of this grammar:

- $!(\mathit{resvsym})$ means any character that is not a *resvsym*.
- $\{\mathit{digit}\}/\{\mathit{char}\}/\{\mathit{expr}\}$ means zero or more *digit*s, *char*s or *expr*s, respectively.

In Ruse, all function or operator calls need to be surrounded by parentheses. To reduce the number of parentheses in the grammar, note the rule

$$\mathit{expr} ::= \mathit{int} \mid \mathit{bool} \mid \mathit{string} \mid \mathit{ident} \mid \text{“(”}\ \mathit{pexpr}\ \text{“)”}$$

What this rule says is that an *expr* is either an *int*, a *bool*, an *ident* (variable name), or an *pexpr* surrounded by parentheses. *pexpr* contains all of the supported operators and keywords.

Below we present the full grammar for Ruse.

$$
\begin{aligned}
\mathit{digit} &::= \text{“\texttt{0}”} \mid \text{“\texttt{1}”} \mid \text{“\texttt{2}”} \mid \text{“\texttt{3}”} \mid \text{“\texttt{4}”} \mid \text{“\texttt{5}”} \mid \text{“\texttt{6}”} \mid \text{“\texttt{7}”} \mid \text{“\texttt{8}”} \mid \text{“\texttt{9}”} \\
\mathit{int} &::= [\text{“}-\text{”}]\ \mathit{digit}\ \{\mathit{digit}\} \\
\mathit{bool} &::= \text{“\#\texttt{t}”} \mid \text{“\#\texttt{f}”} \\
\mathit{string} &::= \text{“ “ ”}\ \{!(\text{“ ” ”})\}\ \text{“ ” ”} \\
\mathit{alpha} &::= \text{“\texttt{a}”} \mid \text{“\texttt{b}”} \mid \ldots \mid \text{“\texttt{z}”} \mid \text{“\texttt{A}”} \mid \text{“\texttt{B}”} \mid \ldots \mid \text{“\texttt{Z}”} \\
\mathit{resvsym} &::= \text{“ (”} \mid \text{“) ”} \mid \text{“ [”} \mid \text{“] ”} \mid \text{“ \{ ”} \mid \text{“ \} ”} \mid \text{“ ” ”} \mid \text{“ \textbackslash{} ”} \mid \text{“ , ”} \mid \text{“ ‘ ”} \mid \text{“ ; ”} \mid \text{“ \# ”} \mid \text{“ | ”} \\
\mathit{ident} &::= \mathit{alpha}\ \{\ !(\mathit{revsym})\ \} \\
\mathit{expr} &::= \mathit{int} \mid \mathit{bool} \mid \mathit{string} \mid \mathit{ident} \mid \text{“(”}\ \mathit{pexpr}\ \text{“)”} \\
\mathit{pexpr} &::= \text{“ + ”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \text{“ }-\text{ ”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \text{“ }*\text{ ”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \text{“\texttt{or}”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \text{“\texttt{and}”}\ \mathit{expr}_1\ \mathit{expr}_2 \\
&\quad \mid \text{“\texttt{not}”}\ \mathit{expr} \mid \text{“\texttt{eq}?”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \text{“ }<\text{ ”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \text{“ }>\text{ ”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \text{“\texttt{nil}?”}\ \mathit{expr} \\
&\quad \mid \text{“\texttt{list}”}\ \mathit{expr}\ \{\mathit{expr}\} \mid \text{“ ”} \mid \text{“\texttt{car}”}\ \mathit{expr} \mid \text{“\texttt{cdr}”}\ \mathit{expr} \mid \text{“\texttt{cons}”}\ \mathit{expr}_1\ \mathit{expr}_2 \mid \mathit{expr}_1\ \mathit{expr}_2 \\
&\quad \mid \text{“\texttt{if}”}\ \mathit{expr}_1\ \mathit{expr}_2\ \mathit{expr}_3 \mid \text{“\texttt{lambda}”}\ \mathit{ident}\ \mathit{expr} \mid \text{“\texttt{define}”}\ \mathit{ident}\ \mathit{expr}
\end{aligned}
$$

A few notes about some of the cases:

- The `list` keyword is used to construct *non-empty* lists; the empty list is written as just “()”.
- A string is zero-or-more non-double-quote characters, surrounded by two double quotes.

Spacing is difficult to specify precisely. Follow these general guidelines:

1. A left-parentheses “(” can be followed by *zero*-or-more spaces.
2. A right-parentheses “)” can be preceded by *zero*-or-more spaces.

3. Every keyword (e.g., `not`) and operation symbol (e.g., <) must be followed by *one*-or-more spaces.

4. A space is *not* needed in expressions like -42, since $-$ is not an operation symbol.

5. There must be *one*-or-more spaces between any two expressions.

2 Semantics for Ruse

Below, you can find the operational semantics for the Ruse programming language. Unlike the semantics that you wrote in the first written assignment, here we use *big-step semantics*: the statement $e \Downarrow v$ asserts that a Ruse expression e leads to the final value v. Furthermore, each expression has the potential to report an error during evaluation. Throughout, we use $\mathcal{E}$ to represent some kind of error. In the code, an error is represented by `Left` *str*, where *str* is a string describing the error.

Handling Evaluation Errors

Below are the operational semantics for the `Plus` operation. We define how `Plus` should behave normally, when the first expression evaluates to an error, and when the second expression evaluates to an error. In general, if an error occurs, the first error should be returned. For example, in the Plus-SError rule, the error created by evaluating e_2 should only be returned if e_1 evaluates normally. While we will not write similar rules for all types of expressions, your evaluator should handle errors in this fashion.

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{\texttt{Plus}(e_1, e_2) \Downarrow n}\ \textsc{Plus} \qquad \frac{e_1 \Downarrow \mathcal{E}}{\texttt{Plus}(e_1, e_2) \Downarrow \mathcal{E}}\ \textsc{Plus-Err1} \qquad \frac{e_1 \Downarrow n \qquad e_2 \Downarrow \mathcal{E}}{\texttt{Plus}(e_1, e_2) \Downarrow \mathcal{E}}\ \textsc{Plus-Err2}$$

Type Errors

These semantics now allow for errors to be threaded throughout an expression, but do not allow for new errors to be generated. Since Ruse is a dynamically-typed language, there can be type errors during runtime, such as trying to add an integer to a Boolean.

To describe this behavior concisely, for each type of expression, we will provide the rule for its normal behavior paired with one-or-more "fallback" rules which will generate a new error. For example, for `Plus`, the rules would be:

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{\texttt{Plus}(e_1, e_2) \Downarrow n}\ \textsc{Plus} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \mathcal{E} = \text{“}\texttt{Add on non-numeric}\text{”}}{\texttt{Plus}(e_1, e_2) \Downarrow \mathcal{E}}\ \textsc{Plus-Err}$$

In general, we won't list the equivalent Plus-Err rule for every expression. Instead, your evaluator should return a new error if the type of the component expressions is not the right type; the starter code includes the error strings you should use each case.

Values

Ruse values are programs that are completed. We use the letter v to stand for any kind of value. In the big-step semantics, values evaluate to themselves:

$$\frac{}{v \Downarrow v}\ \textsc{Val}$$

Ruse has the following kinds of values:

- Numeric values (n): these are numbers, like "42" and "-1".

- Boolean values (b): these are written "#`t`" and "#`f`".
- Function values (f): these are written `Lam`(e)
- Lists of values (vs): these are written `List`$[v_1, \ldots, v_n]$

In our semantics below, we use the letter in parentheses to represent any value from that kind of value; for instance, n represents any numeric value.

Arithmetic Operators

There are three arithmetic operators: `Plus`, `Subt`, and `Mult`. They have similar operational semantics:

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{\texttt{Plus}(e_1, e_2) \Downarrow n}\ \textsc{Plus} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 - n_2}{\texttt{Subt}(e_1, e_2) \Downarrow n}\ \textsc{Subt} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 \cdot n_2}{\texttt{Mult}(e_1, e_2) \Downarrow n}\ \textsc{Mult}$$

Boolean Operators

Here we will define the operational semantics for `And`, `Or`, `Not`, `IsEq`, `IsLt`, and `IsGt`. Note that `And` and `Or` "short-circuit", that is, they *don't* always evaluate both of their arguments to values. In Ruse, just like in Scheme, true and false are written "#`t`" and "#`f`".

$$\frac{e_1 \Downarrow \text{“\#}\texttt{t}\text{”} \qquad e_2 \Downarrow b_2}{\texttt{And}(e_1, e_2) \Downarrow b_2}\ \textsc{And-True} \qquad \frac{e_1 \Downarrow \text{“\#}\texttt{f}\text{”}}{\texttt{And}(e_1, e_2) \Downarrow \text{“\#}\texttt{f}\text{”}}\ \textsc{And-False}$$

$$\frac{e_1 \Downarrow \text{“\#}\texttt{t}\text{”}}{\texttt{Or}(e_1, e_2) \Downarrow \text{“\#}\texttt{t}\text{”}}\ \textsc{Or-True} \qquad \frac{e_1 \Downarrow \text{“\#}\texttt{f}\text{”} \qquad e_2 \Downarrow b_2}{\texttt{Or}(e_1, e_2) \Downarrow b_2}\ \textsc{Or-False}$$

$$\frac{e \Downarrow \text{“\#}\texttt{t}\text{”}}{\texttt{Not}(e) \Downarrow \text{“\#}\texttt{f}\text{”}}\ \textsc{Not-True} \qquad \frac{e \Downarrow \text{“\#}\texttt{f}\text{”}}{\texttt{Not}(e) \Downarrow \text{“\#}\texttt{t}\text{”}}\ \textsc{Not-False}$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad b = (n_1 \ \textbf{equals}\ n_2)}{\texttt{IsEq}(e_1, e_2) \Downarrow b}\ \textsc{IsEq}$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad b = (n_1 < n_2)}{\texttt{IsLt}(e_1, e_2) \Downarrow b}\ \textsc{IsLt} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad b = (n_1 > n_2)}{\texttt{IsGt}(e_1, e_2) \Downarrow b}\ \textsc{IsGt}$$

If-Then-Else Statements

Here we give the semantics for `Ifte`. In Ruse, if-then-else statements are lazy.

$$\frac{e \Downarrow \text{“\#}\texttt{t}\text{”} \qquad e_1 \Downarrow v}{\texttt{Ifte}(e, e_1, e_2) \Downarrow v}\ \textsc{If-True} \qquad \frac{e \Downarrow \text{“\#}\texttt{f}\text{”} \qquad e_2 \Downarrow v}{\texttt{Ifte}(e, e_1, e_2) \Downarrow v}\ \textsc{If-False}$$

In particular, note that we *do not* evaluate the branch that is not taken.

List Operations

Here we give the semantics for `List`, `Cons`, `Car`, `Cdr`, and `IsNil`.

$$\frac{e_1 \Downarrow v_1 \qquad \cdots \qquad e_n \Downarrow v_n}{\texttt{List}([e_1, \ldots, e_n]) \Downarrow [v_1, \ldots, v_n]}\ \textsc{List} \qquad \frac{e \Downarrow v \qquad es \Downarrow [v_1, \ldots, v_n] \qquad vs = [v, v_1, \ldots, v_n]}{\texttt{Cons}(e, es) \Downarrow vs}\ \textsc{Cons}$$

$$\frac{e \Downarrow [v_1, \ldots, v_n]}{\texttt{Car}(e) \Downarrow v_1}\ \textsc{Car} \qquad \frac{e \Downarrow [\,] \qquad \mathcal{E} = \text{“}\texttt{car on empty list}\text{”}}{\texttt{Car}(e) \Downarrow \mathcal{E}}\ \textsc{Car-Empty}$$

$$\frac{e \Downarrow [v_1, v_2, \ldots, v_n]}{\texttt{Cdr}(e) \Downarrow [v_2, \ldots, v_n]}\ \textsc{Cdr} \qquad \frac{e \Downarrow [\,] \qquad \mathcal{E} = \text{“}\texttt{cdr on empty list}\text{”}}{\texttt{Cdr}(e) \Downarrow \mathcal{E}}\ \textsc{Cdr-Empty}$$

$$\frac{e \Downarrow [\,]}{\texttt{IsNil}(e) \Downarrow \text{“}\#\texttt{t}\text{”}}\ \textsc{IsNil-True} \qquad \frac{e \Downarrow [v_1, \ldots, v_n]}{\texttt{IsNil}(e) \Downarrow \text{“}\#\texttt{f}\text{”}}\ \textsc{IsNil-False}$$

Note that `Car` and `Cdr` can generate *two* kinds of new errors: one error if the argument is not a list, and another error if the argument is an empty list. Also note that we left off the `List` tag from the right-sides of the step relations; for instance, $[v_1, \ldots, v_n]$ represents $\texttt{List}([v_1, \ldots, v_n])$, and $[\,]$ represents $\texttt{List}([])$.

Functions

Here we give the semantics for `App` and `Rec`. We let `x` be the default variable for the argument of a function f and `f` be the default variable for the recursive call in a `Rec` expression.

$$\frac{e_1 \Downarrow \lambda\texttt{x}.\ e \qquad e_2 \Downarrow v \qquad e[\texttt{x} \mapsto v] \Downarrow v'}{\texttt{App}(e_1, e_2) \Downarrow v'}\ \textsc{App} \qquad \frac{e[\texttt{f} \mapsto \texttt{Rec}(e)] \Downarrow v}{\texttt{Rec}(e) \Downarrow v}\ \textsc{Rec}$$

In the homework code, we have encoded functions so that the name of bound variables (e.g., `x` and `f`) is not mentioned explicitly. To substitute an expression into another expression, you should use the substitution function `subst` that we have provided for you.