

1 Syntax

$$\begin{aligned}
S = & \mathbf{pure} \ E \\
& | \ x = S_1; S_2 \\
& | \ f \ E^* \\
& | \ \mathbf{fail} \\
& | \ S_1 \parallel S_2 \\
& | \ S_1 \triangleleft S_2 \\
& | \ \mathbf{get} \ E \\
& | \ \mathbf{peek} \\
& | \ \mathbf{parse} \ S \ E \\
& | \ \mathbf{case} \ E \ \mathbf{of} \ (P \rightarrow S)^+ \\
E = & \dots \text{language of expressions} \dots
\end{aligned}$$

Figure 1: The Core language

2 Semantics

In this section we present a few different formulations of the semantics of the Core language. Throughout, we use the following notation:

I	The type of streams of tokens (i.e., strings)
X, Y, Z	Refer to elements of I
$\#$	Concatenates members of I
V	The type of semantic values
u, v	Refer to elements of V

The dynamic environments are implicit, except in the relational specification. The notation $\llbracket _ \rrbracket^{x=v}$ indicates the dynamic environment is extended with a new binding of variable x to v .

We won't specify the details of expression language is as it is somewhat orthogonal to the semantics of parsers:

$$\llbracket E \rrbracket : V$$

2.1 Set Semantics

The semantics of a parser is a set of triples (v, X, Y) :

$$\llbracket S \rrbracket : \{(V, I, I)\}$$

If (v, X, Y) is in the semantics of S , then when applied to input $X ++ Y$, S will consume X and produce result v . This formulation allows us to talk about parsers in context. A parser *accepts* an input if it doesn't fail on it:

$$\text{accepts } S \ (X ++ Y) = \exists v. (v, X, Y) \in \llbracket S \rrbracket$$

$$\begin{aligned} \llbracket \mathbf{pure} \ E \rrbracket &= \{(\llbracket E \rrbracket, [], X)\} \\ \llbracket x = S_1; S_2 \rrbracket &= \{(v, X ++ Y, Z) \mid (u, X, Y ++ Z) \leftarrow \llbracket S_1 \rrbracket, (v, Y, Z) \leftarrow \llbracket S_2 \rrbracket^{x=u}\} \\ \llbracket \mathbf{fail} \rrbracket &= \{\} \\ \llbracket S_1 \parallel S_2 \rrbracket &= \llbracket S_1 \rrbracket \cup \llbracket S_2 \rrbracket \\ \llbracket S_1 \triangleleft S_2 \rrbracket &= \llbracket S_1 \rrbracket \cup \{(v, X, Y) \mid (v, X, Y) \leftarrow \llbracket S_2 \rrbracket, \text{not } (\text{accepts } S_1 \ (X ++ Y))\} \\ \llbracket \mathbf{get} E \rrbracket &= \{(X, X, Y) \mid |X| = \llbracket E \rrbracket\} \\ \llbracket \mathbf{peek} \rrbracket &= \{(X, [], X)\} \\ \llbracket \mathbf{parse} \ S \ E \rrbracket &= \{(v, [], Z) \mid (v, X, Y) \leftarrow \llbracket S \rrbracket, \llbracket E \rrbracket = X ++ Y\} \\ \llbracket \mathbf{case} \ E \ \mathbf{of} \ A \rrbracket &= \llbracket \mathbf{select} \ \llbracket E \rrbracket \ A \rrbracket \end{aligned}$$

Figure 2: Set semantics of Core parsers.

Example of a parser that depends on context:

$$S = x = \mathbf{peek}; \mathbf{case} \ x \ \mathbf{of} \ \{\ [] \rightarrow \mathbf{fail}; _ \rightarrow \mathbf{pure} \ () \}$$

This parser accepts the empty string, but only if it is not at the end of the input.

2.2 Semantics as a Relation

This is an alternative presentation of the set semantics.

$$\begin{array}{c}
\text{PURE} \\
\frac{\Gamma \vdash E \rightarrow v}{\Gamma \vdash \mathbf{pure} E \rightarrow v \triangleright [] \cdot X} \\
\\
\text{ADVANCE} \\
\frac{\Gamma \vdash E \rightarrow |X|}{\Gamma \vdash \mathbf{get} E \rightarrow X \triangleright X \cdot Y} \\
\\
\text{LOOK-AHEAD} \\
\frac{}{\Gamma \vdash \mathbf{peek} \rightarrow X \triangleright [] \cdot X} \\
\\
\text{SEQUENCE} \\
\frac{\Gamma \vdash S_1 \rightarrow u \triangleright X \cdot Y ++ Z \quad \Gamma, x = u \vdash S_2 \rightarrow v \triangleright Y \cdot Z}{\Gamma \vdash x = S_1; S_2 \rightarrow v \triangleright X ++ Y \cdot Z} \\
\\
\begin{array}{cc}
\text{UNBIASED-CHOICE-LEFT} & \text{UNBIASED-CHOICE-RIGHT} \\
\frac{\Gamma \vdash S_1 \rightarrow v \triangleright X \cdot Y}{\Gamma \vdash S_1 [] S_2 \rightarrow v \triangleright X \cdot Y} & \frac{\Gamma \vdash S_2 \rightarrow v \triangleright X \cdot Y}{\Gamma \vdash S_1 [] S_2 \rightarrow v \triangleright X \cdot Y}
\end{array} \\
\\
\begin{array}{cc}
\text{BIASED-CHOICE-LEFT} & \text{BIASED-CHOICE-RIGHT} \\
\frac{\Gamma \vdash S_1 \rightarrow v \triangleright X \cdot Y}{\Gamma \vdash S_1 \triangleleft S_2 \rightarrow v \triangleright X \cdot Y} & \frac{\Gamma \vdash S_2 \rightarrow v \triangleright X \cdot Y \quad \Gamma \vdash (X ++ Y) \notin S_1}{\Gamma \vdash S_1 \triangleleft S_2 \rightarrow v \triangleright X \cdot Y}
\end{array} \\
\\
\text{NESTED-PARSER} \\
\frac{\Gamma \vdash E \rightarrow (X ++ Y ++ Z) \quad \Gamma \vdash S \rightarrow v \triangleright X \cdot Y}{\Gamma \vdash \mathbf{parse} S E \rightarrow v \triangleright [] \cdot Z} \\
\\
\text{CASE} \\
\frac{\Gamma \vdash E \rightarrow u \quad \Gamma \vdash \mathbf{select} u A \rightarrow v \triangleright X \cdot Y}{\Gamma \vdash \mathbf{case} E \mathbf{of} A \rightarrow v \triangleright X \cdot Y}
\end{array}$$

Figure 3: $\Gamma \vdash S \rightarrow v \triangleright X \cdot Y$ describes the behavior of parser S in dynamic environment Γ . When applied to the input $X ++ Y$, S will consume X and produce semantic value v .

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\Gamma \vdash X \notin \mathbf{fail} \\
\\
\text{UNBIASED-MISMATCH} \\
\frac{\Gamma \vdash X \notin S_1 \quad \Gamma \vdash X \notin S_2}{\Gamma \vdash X \notin S_1 \parallel S_2} \\
\\
\text{NOT-FRONT} \\
\frac{\Gamma \vdash X \notin S_1}{\Gamma \vdash X \notin x = S_1; S_2} \\
\\
\text{NOT-NESTED} \\
\frac{\Gamma \vdash E \rightarrow v \quad \Gamma \vdash v \notin P}{\Gamma \vdash X \notin \mathbf{parse} P E} \\
\\
\text{TOO-SHORT} \\
\frac{\Gamma \vdash E \rightarrow v \quad |X| < v}{\Gamma \vdash X \notin \mathbf{get} E} \\
\\
\text{BIASED-MISMATCH} \\
\frac{\Gamma \vdash X \notin S_1 \quad \Gamma \vdash X \notin S_2}{\Gamma \vdash X \notin S_1 \triangleleft S_2} \\
\\
\text{NOT-BACK} \\
\frac{\Gamma \vdash S_1 \rightarrow v \triangleright X \cdot Y \quad \Gamma, x = v \vdash Y \notin S_2}{\Gamma \vdash (X ++ Y) \notin x = S_1; S_2} \\
\\
\text{NO-CASE} \\
\frac{\Gamma \vdash E \rightarrow v \quad \Gamma \vdash X \notin \mathbf{select} v A}{\Gamma \vdash X \notin \mathbf{case} E \mathbf{of} A}
\end{array}$$

Figure 4: $\Gamma \vdash X \notin S$ asserts that X is not accepted by S in the sense described before.

3 Semantics as a State Transformer

Another way to give semantics is to model them as a state transformer:

$$\llbracket S \rrbracket : I \rightarrow \{(V, I)\}$$

$$\begin{aligned}
\llbracket \mathbf{pure} E \rrbracket X &= \{(\llbracket E \rrbracket, X)\} \\
\llbracket x = S_1; S_2 \rrbracket X &= \{(v, Z) \mid (u, Y) \leftarrow \llbracket S_1 \rrbracket X, (v, Z) \leftarrow \llbracket S_2 \rrbracket^{x=u} Y\} \\
\llbracket \mathbf{fail} \rrbracket X &= \{\} \\
\llbracket S_1 \parallel S_2 \rrbracket X &= \llbracket S_1 \rrbracket X \cup \llbracket S_2 \rrbracket X \\
\llbracket S_1 \triangleleft S_2 \rrbracket X &= \begin{cases} \llbracket S_1 \rrbracket X & \text{if } \llbracket S_1 \rrbracket X \neq \emptyset \\ \llbracket S_2 \rrbracket X & \text{otherwise} \end{cases} \\
\llbracket \mathbf{get} E \rrbracket X &= \begin{cases} \{(Y, Z)\} & \text{if } |Y| = \llbracket E \rrbracket \wedge X = Y ++ Z \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket \mathbf{peek} \rrbracket X &= \{(X, X)\} \\
\llbracket \mathbf{parse} S E \rrbracket X &= \{(v, X) \mid (v, Y) \leftarrow \llbracket S \rrbracket \llbracket E \rrbracket\} \\
\llbracket \mathbf{case} E \mathbf{of} A \rrbracket X &= \llbracket \mathbf{select} \llbracket E \rrbracket A \rrbracket X
\end{aligned}$$

Figure 5: State transformer semantics of Core parsers.

4 Set vs. State Transformer Semantics

$$(v, X, Y) \in \llbracket S \rrbracket^{\text{set}} \iff (v, Y) \in \llbracket S \rrbracket^{\text{fun}} (X ++ Y)$$

Using state transformers is a more powerful abstraction than what is expressible in Core. In particular, consider an extension of Core that allows for direct stream manipulation, **setStream** E , which returns no interesting semantic value, but modifies the stream that we are parsing. Such a construct is readily expressible using the state transformers semantics:

$$\llbracket \text{setStream } E \rrbracket^{\text{fun}} X = \{((\text{()}, \llbracket E \rrbracket))\}$$

We cannot, however, express such a parser using the set-based semantics, because in this formalism, parsers declare constraints on a global stream, but they cannot change the actual stream. One attempt to define the semantics of such a construct could be:

$$\llbracket \text{setStream } E \rrbracket^{\text{set}} = \{((\text{()}, \llbracket E \rrbracket))\}$$

This, however, is not correct because instead of changing the stream, we are making a look-ahead assertion about what the stream should be. Thus, we'll reject any inputs that do not match E , which is quite different than the intended semantics, which is a parser that never fails but modifies the input. As a concrete example, consider **setStream** "a":

$$\begin{aligned} \llbracket \text{setStream "a"} \rrbracket^{\text{fun}} X &= \{((\text{()}, \text{"a"}))\} \\ \llbracket \text{setStream "a"} \rrbracket^{\text{set}} &= \{((\text{()}, \text{"", "a"}))\} \end{aligned}$$

$$\begin{aligned} (\text{()}, \text{"a"}) &\in \llbracket \text{setStream "a"} \rrbracket^{\text{fun}} (\text{"" ++ "b"}) \\ (\text{()}, \text{"", "b"}) &\notin \llbracket \text{setStream "a"} \rrbracket^{\text{set}} \end{aligned}$$