Lecture Notes on Abstract Syntax

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 3
September 2, 2003

Grammars, as we have discussed them so far, define a formal language as a set of strings. We refer to this as the *concrete syntax* of a language. While this is necessary in the complete definition of a programming language, it is only the beginning. We further have to define at least the static semantics (via typing rules) and the dynamic semantics (via evaluation rules). Then we have to reason about their relationship to establish, for example, type soundness. Giving such definitions and proofs on strings is extremely tedious and inappropriate; instead we want to give it a more abstract form of representation. We refer to this layer of representation as the *abstract syntax* of a language. An appropriate representation vehicle are *terms* [Ch. 1.2.1].

Given this distinction, we can see that parsing is more than simply recognizing if a given string lies within the language defined by a grammar. Instead, parsing in our context should translate a string, given in concrete syntax, into an abstract syntax term. The converse problem of printing (or unparsing) is to translate an abstract syntax term into a string representation. While the grammar formalism is somewhat unwieldy when it comes to specifying the translation into abstract syntax, we see that the mechanism of judgments is quite robust and can specify both parsing and unparsing quite cleanly.

We begin by reviewing the arithmetic expression language in its concrete [Ch. 3] and abstract [Ch. 4.1] forms. First, the grammar in its unambigous form.[1] We implement here the decision that addition and multiplication should be left-associative (so `1+2+3` is parsed as `(1+2)+3`) and that

[1]We capitalize the non-terminals to avoid confusion when considering both concrete and abstract syntax in the same judgment. Also, the syntactic category of $Terms$ (denoted by T) should not be confused with the terms we use to construct abstract syntax.

multiplication has precedence over addition. Such choices are somewhat arbitrary and dictated by convention rather than any scientific criteria.[2]

$$
\begin{array}{llll}
\textit{Digits} & D & ::= & \mathsf{0} \mid \cdots \mid \mathsf{9} \\
\textit{Numbers} & N & ::= & D \mid N\,D \\
\textit{Expressions} & E & ::= & T \mid E\texttt{+}T \\
\textit{Terms} & T & ::= & F \mid T\texttt{*}F \\
\textit{Factors} & F & ::= & N \mid \texttt{(}E\texttt{)}
\end{array}
$$

Written in the form of five judgments.

$$\frac{}{\mathsf{0}\ \mathsf{D}} \quad \cdots \quad \frac{}{\mathsf{9}\ \mathsf{D}}$$

$$\frac{s\ \mathsf{D}}{s\ \mathsf{N}} \quad \frac{s_1\ \mathsf{N} \qquad s_2\ \mathsf{D}}{s_1\, s_2\ \mathsf{N}}$$

$$\frac{s\ \mathsf{T}}{s\ \mathsf{E}} \quad \frac{s_1\ \mathsf{E} \qquad s_2\ \mathsf{T}}{s_1\texttt{+}s_2\ \mathsf{E}}$$

$$\frac{s\ \mathsf{F}}{s\ \mathsf{T}} \quad \frac{s_1\ \mathsf{T} \qquad s_2\ \mathsf{F}}{s_1\texttt{*}s_2\ \mathsf{T}}$$

$$\frac{s\ \mathsf{N}}{s\ \mathsf{F}} \qquad \frac{s\ \mathsf{E}}{\texttt{(}s\texttt{)}\ \mathsf{F}}$$

The abstract syntax of the language is much simpler. It can be specified in the form of a grammar, where the universe we are working over are terms and not strings. While natural numbers can also be inductively defined in a variety of ways [Ch 1.1.1], we take them here as primitive mathematical objects.

$$
\begin{array}{lll}
\mathsf{nat} & ::= & 0 \mid 1 \mid \cdots \\
\mathsf{expr} & ::= & \texttt{num}(\mathsf{nat}) \mid \texttt{plus}(\mathsf{expr}, \mathsf{expr}) \mid \texttt{times}(\mathsf{expr}, \mathsf{expr})
\end{array}
$$

Presented as two judgments, we have k nat for every natural number k and the following rule for expressions

[2]The grammar given in [Ch. 3.2] is slightly different, since there addition and multiplication are assumed to be right associative.

$$\frac{k\ \mathsf{nat}}{\mathtt{num}(k)\ \mathsf{expr}}$$

$$\frac{t_1\ \mathsf{expr} \qquad t_2\ \mathsf{expr}}{\mathtt{plus}(t_1, t_2)\ \mathsf{expr}}$$

$$\frac{t_1\ \mathsf{expr} \qquad t_2\ \mathsf{expr}}{\mathtt{times}(t_1, t_2)\ \mathsf{expr}}$$

Now we specify the proper relation between concrete and abstract syntax through several simultaneously inductive judgments. Perhaps the easiest way to generate these judgments is to add the corresponding abstract syntax terms to each of the inference rules defining the concrete syntax.

$$\overline{\mathtt{0}\ \mathsf{D} \longleftrightarrow 0\ \mathsf{nat}} \quad \cdots \quad \overline{\mathtt{9}\ \mathsf{D} \longleftrightarrow 9\ \mathsf{nat}}$$

$$\frac{s\ \mathsf{D} \longleftrightarrow k\ \mathsf{nat}}{s\ \mathsf{N} \longleftrightarrow k\ \mathsf{nat}} \qquad \frac{s_1\ \mathsf{N} \longleftrightarrow k_1\ \mathsf{nat} \qquad s_2\ \mathsf{D} \longleftrightarrow k_2\ \mathsf{nat}}{s_1\, s_2\ \mathsf{N} \longleftrightarrow 10k_1 + k_2\ \mathsf{nat}}$$

$$\frac{s\ \mathsf{T} \longleftrightarrow t\ \mathsf{expr}}{s\ \mathsf{E} \longleftrightarrow t\ \mathsf{expr}} \qquad \frac{s_1\ \mathsf{E} \longleftrightarrow t_1\ \mathsf{expr} \qquad s_2\ \mathsf{T} \longleftrightarrow t_2\ \mathsf{expr}}{s_1\mathtt{+}s_2\ \mathsf{E} \longleftrightarrow \mathtt{plus}(t_1, t_2)\ \mathsf{expr}}$$

$$\frac{s\ \mathsf{F} \longleftrightarrow t\ \mathsf{expr}}{s\ \mathsf{T} \longleftrightarrow t\ \mathsf{expr}} \qquad \frac{s_1\ \mathsf{T} \longleftrightarrow t_1\ \mathsf{expr} \qquad s_2\ \mathsf{F} \longleftrightarrow t_2\ \mathsf{expr}}{s_1\mathtt{*}s_2\ \mathsf{T} \longleftrightarrow \mathtt{times}(t_1, t_2)\ \mathsf{expr}}$$

$$\frac{s\ \mathsf{N} \longleftrightarrow k\ \mathsf{nat}}{s\ \mathsf{F} \longleftrightarrow \mathtt{num}(k)\ \mathsf{expr}} \qquad \frac{s\ \mathsf{E} \longleftrightarrow t\ \mathsf{expr}}{\mathtt{(}s\mathtt{)}\ \mathsf{F} \longleftrightarrow t\ \mathsf{expr}}$$

When giving a specification of the form above, we should verify that the basic properties we expect, actually hold. In this case we would like to check that related strings and terms belong to the correct (concrete or abstract, respectively) syntactic classes.

Theorem 1

(i) If $s\ \mathsf{D} \longleftrightarrow k\ \mathsf{nat}$ *then* $s\ \mathsf{D}$ *and* $k\ \mathsf{nat}$.

(ii) If $s\ \mathsf{N} \longleftrightarrow k\ \mathsf{nat}$ *then* $s\ \mathsf{N}$ *and* $k\ \mathsf{nat}$.

(iii) If $s\ \mathsf{E} \longleftrightarrow t\ \mathsf{expr}$ *then* $s\ \mathsf{E}$ *and* $t\ \mathsf{expr}$.

(iv) If $s\ \mathsf{T} \longleftrightarrow t\ \mathsf{expr}$ *then* $s\ \mathsf{T}$ *and* $t\ \mathsf{expr}$.

(v) *If* s $\mathsf{F} \longleftrightarrow t$ expr *then* s F *and* t expr.

Proof: Part (i) follows by cases (there are 10 cases, one for each digit).

Part (ii) follows by rule induction on the given derivation, using (i) in both cases.

Parts (iii), (iv), and (v) follow by simultaneous rule induction on the given derivation, using part (ii) in one case. Overall, there are 6 cases. In each case we can immediately appeal to the induction hypothesis on all subderivations and construct a derivation of the desired judgment from the results. ■

When implementing such a specification, we generally make a commitment as to what is considered our input and what is our output. As motivated above, parsing and unparsing (printing) are specifed by this judgment.

Definition 2 (Parsing)
Given a string s*, find a term* t *such that* s $\mathsf{E} \longleftrightarrow t$ expr *or fail, if no such* t *exists.*

Obvious analogous definitions exist for the other syntactic categories. We can further relate parsing into abstract syntax to our definition of the syntactic categories by ascertaining that if s E then there is an abstract syntax term representing it.

Theorem 3

(i) *If* s D *then there is a* k *with* s $\mathsf{D} \longleftrightarrow k$ nat.

(ii) *If* s N *then there is a* k *with* s $\mathsf{N} \longleftrightarrow k$ nat.

(iii) *If* s E *then there is a* t *with* s $\mathsf{E} \longleftrightarrow t$ expr.

(iv) *If* s T *then there is a* t *with* s $\mathsf{T} \longleftrightarrow t$ expr.

(v) *If* s F *then there is a* t *with* s $\mathsf{F} \longleftrightarrow t$ expr.

Proof: By cases or straightforward rule induction as in the proof of Theorem 1. ■

Now we can refine our notion of ambiguity to take into account the abstract syntax that is constructed. This is slightly more relaxed that requiring the uniqueness of derivations, because different derivations could still lead to the same abstract syntax term.

Definition 4 (Ambiguity of Parsing)
A parsing problem is ambiguous if for a given string s there exist two distinct terms t_1 and t_2 such that $s\ \mathsf{E} \longleftrightarrow t_1\ \mathsf{expr}$ and $s\ \mathsf{E} \longleftrightarrow t_2\ \mathsf{expr}$.

Unparsing is just the reverse of parsing: we are given a term t and have to find a concrete syntax representation for it. Unparsing is usually total (every term can be unparsed) and inherently ambiguous (the same term can be written as several strings). An example of this ambiguity is the insertion of additional redundant parentheses. Therefore, any unparser must use heuristics to choose among different alternative string representations.

Definition 5 (Unparsing)
Given a term t such that $t\ \mathsf{expr}$, find a string s such that $s\ \mathsf{E} \longleftrightarrow t\ \mathsf{expr}$.

The ability to use judgments as the basis for implementation of different tasks is evidence for their flexibility. Often, it is not difficult to "translate" a judgment into an implementation in a high-level language such as ML, although in some cases it might require significant ingenuity and some advanced techniques.

Our little language of arithmetic expressions serves to illustrate various ideas, such as the distinction between concrete syntax and abstract syntax, but it is too simple to exhibit various other phenomena and concepts. One of the most important one is that of a variable, and the notion of variable binding and scope. In order to discuss variables in isolation, we extend our language by a new form of expression to name preliminary results. For example,

$$\texttt{let}\ x = \texttt{2*3}\ \texttt{in}\ x\texttt{+}x\ \texttt{end}$$

should evaluate to 12, but only compute the value of `2*3` once.

First, the concrete syntax, showing only the changed or new cases.

$$\begin{array}{lcll} \textit{Variables} & X & ::= & \textit{(any identifier)} \\ \textit{Factors} & F & ::= & N \mid \texttt{(}E\texttt{)} \mid \texttt{let}\ X = E\ \texttt{in}\ E\ \texttt{end} \mid X \end{array}$$

We ignore here the question what constitutes a legal identifier. Presumably it should avoid keywords (such as `let`, `b`), special symbols, such as `+`, and be surrounded by whitespace. In an actual language implementation a *lexer* breaks the input string into keywords, special symbols, numbers, and identifiers that are the processed by the parser.

The first approach to the abstract syntax would be to simply introduce a new abstract syntactic category of *variable* [Ch. 5.1] and a new operator `let` with three arguments, $\texttt{let}(x, e_1, e_2)$, where x is a variable and e_1 and e_2 are terms representing expressions. Furthermore, we allow an occurrence of a variable x as a term. However, this approach does not clarify which occurrences of a variable are *binding occurrences*, and to which binder a variable occurrence refers. For example, to see that

$$\texttt{let}\ x = \texttt{1}\ \texttt{in}\ \texttt{let}\ x = x\texttt{+1}\ \texttt{in}\ x\texttt{+}x\ \texttt{end}\ \texttt{end}$$

evaluates to 4, we need to know which occurrences of x refer to which values. Rules for scope resolution [Ch. 5.1] dictate that it should be interpreted the same as

$$\texttt{let}\ x_1 = \texttt{1}\ \texttt{in}\ \texttt{let}\ x_2 = x_1\texttt{+1}\ \texttt{in}\ x_2\texttt{+}x_2\ \texttt{end}\ \texttt{end}$$

where there is no longer any potential ambiguity. That is, the scope of the variable x in

$$\texttt{let}\ x = s_1\ \texttt{in}\ s_2\ \texttt{end}$$

is s_2 but not s_1.

A uniform technique to encode the information about the scope of variables is called *higher-order abstract syntax* [Ch. 5]. We add to our language of terms a construct $x.t$ which binds x in the term t. Every occurrence of x in t that is not shadowed by another binding $x.t'$, refers to the shown top-level abstraction. Such variables are a new primitive concept, and, in particular, a variable can be used as a term (in addition to the usual operator-based terms). We would extend our judgment relating concrete and abstract syntax by

$$\frac{x\ \mathsf{X} \qquad s_1\ \mathsf{E} \longleftrightarrow t_1\ \mathsf{expr} \qquad s_2\ \mathsf{E} \longleftrightarrow t_2\ \mathsf{expr}}{\texttt{let}\ x = s_1\ \texttt{in}\ s_2\ \texttt{end} \longleftrightarrow \texttt{let}(t_1, x.t_2)\ \mathsf{expr}} \qquad \frac{x\ \mathsf{X}}{x\ \mathsf{E} \longleftrightarrow x\ \mathsf{expr}}$$

and allow for expressions

$$\frac{}{x\ \mathsf{expr}} \qquad \frac{t_1\ \mathsf{expr} \qquad t_2\ \mathsf{expr}}{\texttt{let}(t_1, x.t_2)\ \mathsf{expr}}$$

Note that we translate an identifier x to an identically named variable x in higher-order abstract syntax. Moreover, we view variables in higher-order abstract syntax as a new kind of term, so we do not check explicitly the x's are in fact variables—it is implied that they are.

We emphasize that the laws for scope resolution of `let`-expressions are directly encoded in the higher-order abstract representation. We investigate the laws underlying such representations in Lecture 4 [Ch. 5.3].

We can formulate the language of abstract syntax for arithmetic expressions in a more compact notation as a grammar.

$$
\begin{array}{lcl}
\mathsf{nat} & ::= & 0 \mid 1 \mid \cdots \\
\mathsf{expr} & ::= & \mathtt{num}(\mathsf{nat}) \mid \mathtt{plus}(\mathsf{expr}, \mathsf{expr}) \mid \mathtt{times}(\mathsf{expr}, \mathsf{expr}) \\
 & & \mid x \mid \mathtt{let}(\mathsf{expr}, x.\mathsf{expr})
\end{array}
$$

As a concrete example, consider the string

$$\mathtt{let}\ x_1 = 1\ \mathtt{in}\ \mathtt{let}\ x_2 = x_1 \mathtt{+1}\ \mathtt{in}\ x_2 \mathtt{+} x_2\ \mathtt{end}\ \mathtt{end}$$

which, in abstract syntax, would be represented as

$$\mathtt{let}(\mathtt{num}(1), x_1.\mathtt{let}(\mathtt{plus}(x_1, \mathtt{num}(1)), x_2.\mathtt{plus}(x_2, x_2)))$$