15-312 Lecture on Concrete and Abstract Syntax

Solution to Exercise 1

In a deductive system containing the rules for $_\ \mathsf{nat}$ and $\mathsf{sum}(_,_,_)$:

$$\frac{}{\mathsf{z}\ \mathsf{nat}}\ \mathbf{z_nat} \qquad \frac{n\ \mathsf{nat}}{\mathsf{s}\ n\ \mathsf{nat}}\ \mathbf{s_nat}$$

$$\frac{n\ \mathsf{nat}}{\mathsf{sum}(\mathsf{z}, n, n)}\ \mathbf{sum_z} \qquad \frac{\mathsf{sum}(m, n, p)}{\mathsf{sum}(\mathsf{s}\ m, n, \mathsf{s}\ p)}\ \mathbf{sum_s}$$

prove that:

> *If* $\mathcal{D} :: \mathsf{sum}(m, n, p)$*, then there exists derivations* $\mathcal{D}_m :: m\ \mathsf{nat}$*,* $\mathcal{D}_n :: n\ \mathsf{nat}$ *and* $\mathcal{D}_p :: p\ \mathsf{nat}$*,*

Proof: The proof proceeds by induction on the structure of the given derivation $\mathcal{D}$. There are two cases to examine:

1. Case:

$$\mathcal{D} = \frac{\begin{matrix}\mathcal{D}'_n \\ n'\ \mathsf{nat}\end{matrix}}{\mathsf{sum}(\mathsf{z}, n', n')}\ \mathbf{sum_z}$$

   Then, it must be the case that $m = \mathsf{z}$, $n = n'$ and $p = n'$. We need to build derivations of $m\ \mathsf{nat}$, $n\ \mathsf{nat}$ and $p\ \mathsf{nat}$. These are easily obtained as follows:

$$\mathcal{D}_m = \frac{}{\mathsf{z}\ \mathsf{nat}}\ \mathbf{z_nat} \qquad \mathcal{D}_n = \mathcal{D}'_n \qquad \mathcal{D}_p = \mathcal{D}'_n$$

   This concludes this case of the proof.

2. Case:

$$\frac{\begin{matrix}\mathcal{D}' \\ \mathsf{sum}(m', n', p')\end{matrix}}{\mathsf{sum}(\mathsf{s}\ m', n', \mathsf{s}\ p')}\ \mathbf{sum_s}$$

Then, it must be the case that $m = \mathsf{s}\, m'$, $n = n'$ and $p = \mathsf{s}\, p'$. Since we have a derivation $\mathcal{D}'$ of $\mathsf{sum}(m', n', p')$, by induction hypothesis on $\mathcal{D}'$, we can assume that there exist derivations $\mathcal{D}'_m :: m'\ \mathsf{nat}$, $\mathcal{D}'_n :: n'\ \mathsf{nat}$ and $\mathcal{D}'_p :: p'\ \mathsf{nat}$.

We need to build derivations of $m\ \mathsf{nat}$, $n\ \mathsf{nat}$ and $p\ \mathsf{nat}$. We obtain them as follows:

$$\mathcal{D}_m = \cfrac{\begin{matrix}\mathcal{D}'_m \\ m'\ \mathsf{nat}\end{matrix}}{\mathsf{s}\, m'\ \mathsf{nat}}\ \mathbf{s_nat} \qquad \mathcal{D}_n = \mathcal{D}'_n \qquad \mathcal{D}_p = \cfrac{\begin{matrix}\mathcal{D}'_p \\ p'\ \mathsf{nat}\end{matrix}}{\mathsf{s}\, p'\ \mathsf{nat}}\ \mathbf{s_nat}$$

This concludes this case of the proof.

This concludes the proofs since we have proved each case. □

Solution to Exercise 2

In the following deductive system for transition sequences:

$$\cfrac{}{s \mapsto^* s}\ \mathbf{id} \qquad \cfrac{s \mapsto s' \quad s' \mapsto^* s''}{s \mapsto^* s''}\ \mathbf{it_r}$$

show that the following rule is admissible:

$$\cfrac{s \mapsto^* s' \quad s' \mapsto s''}{s \mapsto^* s''}\ \mathbf{it_l}$$

Is it derivable?

Let us rephrase this question as a property. Prove that

For every derivations $\mathcal{D} :: s_1 \mapsto^* s_2$ and $\mathcal{E} :: s_2 \mapsto s_3$, there exists a derivation $\mathcal{F} :: s_1 \mapsto^* s_3$

(We have renamed the states for clarity.)

Proof: The proof proceeds by induction on the structure of derivation $\mathcal{D}$. (Note that we have not given any rule for the judgment $s \mapsto s'$ and therefore we have no idea what a derivation of $\mathcal{E}$ looks like. For this reason, we cannot proceed by induction on $\mathcal{E}$). There are two cases to examine, one for each rule defining the judgment $s \mapsto^* s'$:

1. Case:

$$\mathcal{D} = \cfrac{}{s \mapsto^* s}\ \mathbf{id}$$

where $s_1 = s_2 = s$.

Since the derivation $\mathcal{E} :: s \mapsto s_3$ is given to us (recall that $s_2 = s$), we build the desired derivation $\mathcal{F}$ as follows:

$$\mathcal{F} = \cfrac{\begin{matrix}\mathcal{E} \\ s \mapsto s_3\end{matrix} \quad \cfrac{}{s_3 \mapsto^* s_3}\ \mathbf{id}}{s \mapsto^* s_3}\ \mathbf{it_r}$$

Note how we are using $\mathcal{E}$ on the left of the construction for $\mathcal{F}$ while it would appear on the right in the admissible rule. Note also how we needed to reconstruct from scratch the right subderivation of rule **lt_r**.

2. Case:

$$\mathcal{D} = \cfrac{\begin{array}{cc}\mathcal{D}' & \mathcal{D}'' \\ s \mapsto s' & s' \mapsto^* s''\end{array}}{s \mapsto^* s''}\ \textbf{it_r}$$

where $s_1 = s$, $s_2 = s''$ and s' is some intermediate state that must exist for this rule to have been applied.

Since $\mathcal{D}''$ is smaller than $\mathcal{D}$, we can apply the induction hypothesis to $\mathcal{D}''$ and $\mathcal{E}$ and this allows us to postulate the existence of a derivation $\mathcal{F}'' :: s' \mapsto^* s_3$. We can then combine derivations $\mathcal{D}'$ and $\mathcal{F}''$ using rule **it_r** into the desired derivation $\mathcal{F}$ as follows:

$$\mathcal{F} = \cfrac{\begin{array}{cc}\mathcal{D}' & \mathcal{F}'' \\ s \mapsto s' & s' \mapsto^* s_3\end{array}}{s \mapsto^* s_3}\ \textbf{it_r}$$

which is what we wanted since $s_1 = s$.

This concludes the proof of this statement, and this also allows us to conclude that rule

$$\frac{s \mapsto^* s' \quad s' \mapsto s''}{s \mapsto^* s''}\ \textbf{it_l}$$

is admissible. □

To understand why rule **it_l** is not derivable, note the shape of a proof of $s \mapsto s'$ using rules **id** and **it_r**: it is a tree that is completely unbalanced to the right with all the left branches consisting of subderivations of the judgment $_ \mapsto _$. So, every derivable rule may have premises for $_ \mapsto _$ on the left but the rightmost premise will have to be for the $_ \mapsto^* _$ judgment.

Instead, a derivation that uses **it_l** has a subderivation for the $_ \mapsto _$ judgment as its rightmost premise. For this reason, it cannot be derived from rules **id** and **it_r**. Note that it builds a derivation that grows left. Indeed, if only **id** and **it_l** are ever used, the resulting derivation tree will be completely unbalanced toward the left.

To reinforce these concepts, prove that the following deductive system is equivalent to :

$$\frac{}{s \mapsto^* s}\ \textbf{id}' \qquad \frac{s \mapsto s'}{s \mapsto^* s'}\ \textbf{it}'\textbf{_step} \qquad \frac{s \mapsto^* s' \quad s' \mapsto^* s''}{s \mapsto^* s''}\ \textbf{it}'\textbf{_lr}$$

(Note that both premises of **it′_lr** use the transition sequence — not step — judgment.)

Strings from First Principles

Given an alphabet Σ, we define the judgment c char_Σ to indicate that c is a character in Σ. It is extensionally defined by giving one rule for each $c \in \Sigma$:

$$\frac{}{c\ \mathsf{char}_\Sigma}\ \mathbf{c}$$

Then, strings over Σ are just lists of characters with ϵ as the empty string (corresponding to the empty list of characters) and the constructor $_ \cdot _$, where $c \cdot s$ indicates the extension of string s with character c (on the left):

$$\frac{}{\epsilon\ \mathsf{string}_\Sigma}\ \mathbf{str_}\epsilon \qquad \frac{c\ \mathsf{char}_\Sigma \quad s\ \mathsf{string}_\Sigma}{c \cdot s\ \mathsf{string}_\Sigma}\ \mathbf{str_}\cdot$$

From now on, we will assume that the alphabet Σ is fixed, and omit it as a subscript of these judgments.

Traversing strings left to right one character at a time is rather tedious. We'd like to manipulate strings as a concatenation of substrings. Let's define the judgment $s\ \mathsf{as}\ s_1\hat{}s_2$ that splits a string s as the concatenation of two string s_1 and s_2 (or dually builds s as the concatenation of s_1 and s_2):

$$\frac{s\ \mathsf{string}}{s\ \mathsf{as}\ \epsilon\hat{}s}\ \hat{}\mathbf{_r} \qquad \frac{s\ \mathsf{as}\ s_1\hat{}s_2}{c \cdot s\ \mathsf{as}\ (c \cdot s_1)\hat{}s_2}\ \hat{}\mathbf{_l}$$

Note that the mode of these rules is $(\exists!, \forall, \forall)$, thus implementing a concatenation function. However, if we think about them as splitting a given string, they are non-deterministic because a string can be split in many ways (it also has mode $(\forall, \exists, \exists)$).

Lexing

The following grammar provides the raw concrete syntax for simple arithmetic expressions:

Spaces	␣	::=	ϵ | ␣<space> | ␣<tab> | ␣<new line>
Digits	D	::=	`0` | `1` | ... | `9`
Numbers	N	::=	D | $D\,N$
Expressions	E	::=	␣N␣ | E␣`++`␣E | E␣`*`␣E

We have written addition as ++ rather than + for illustration purposes.

Raw concrete syntax is what we type on a computer and write on paper, but it is contains a number of distracting details when we want to reason about a language as a mathematical object. We will eventually rely on *terms*, or *abstract syntax trees*, for this purpose.

A first step in this direction is to isolate the tokens that constitute a phrase, without bothering yet about the tree structure. We want in particular to ignore spaces, interpret numbers as numbers rather than sequences of digits, and give composite keywords (for

example ++ above) an atomic representation. *Lexing* will reduce that grammar to this one:

$$\textit{Expressions} \quad E \quad ::= \quad \mathsf{num}[n] \ \mid \ E + E \ \mid \ E * E$$

Here, $\mathsf{num}[n]$, $+$ and $*$ are tokens, where n is a natural number (in decimal notation for simplicity). We used a different font to distinguish them from characters. We will construct a token stream as a string over these three types of tokens. For convenience, we will build them left-to-right rather than right-to-left as we did earlier — this is done in the same way and we still use ϵ and $\cdot$ as our constructors. We will also use ^ for token stream concatenation (or splitting).

We will now implement lexing as a series of judgments, that model the way an actual lexer works, although in a very simplified and specialized way. We can do so in many ways. We will use two judgments:

$t \ \triangleright \ s \ \mathsf{lex_to} \ t'$ Given token prefix t, string s lexes to tokens t'

$t \ \triangleright_n \ s \ \mathsf{lex_to} \ t'$ Given token prefix t and number n, string s lexes to tokens t'

The first argument of these judgments is used as an accumulator where tokens are stored up to the point where the input string has been totally consumed. It is then output as the third argument. We build a token stream for a concrete string s by attempting to find a derivation for the judgment $\epsilon \ \triangleright \ s \ \mathsf{lex_to} \ t$: if such a derivation exists, then the object t built along the way is a token stream for s. Otherwise, no token stream for s exists.

Although they are mutually recursive, we will now give the defining rules one judgment at a time:

$$\frac{}{t \ \triangleright \ \epsilon \ \mathsf{lex_to} \ t}\ \mathbf{lex_\epsilon} \qquad \frac{t \ \triangleright_d \ s \ \mathsf{lex_to} \ t'}{t \ \triangleright \ d \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex_d}$$

$$\frac{t \ \triangleright \ s \ \mathsf{lex_to} \ t'}{t \ \triangleright \ _ \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex__} \qquad \frac{t \cdot + \ \triangleright \ s \ \mathsf{lex_to} \ t'}{t \ \triangleright \ \texttt{+} \cdot \texttt{+} \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex_+} \qquad \frac{t \cdot * \ \triangleright \ s \ \mathsf{lex_to} \ t'}{t \ \triangleright \ \texttt{*} \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex_*}$$

Rule $\mathbf{lex_\epsilon}$ returns the accumulated token stream when the input string is empty. When seeing a digit d, rule $\mathbf{lex_d}$ passes it to the second judgment (below), which will convert it into a number once all of its digits have been collected. Rule $\mathbf{lex__}$ ignores a space. Rules $\mathbf{lex_+}$ and $\mathbf{lex_*}$ recognize the strings ++ and * respectively and extend the token stream with the tokens $+$ and $*$ respectively.

$$\frac{t \ \triangleright_{n'} \ s \ \mathsf{lex_to} \ t' \quad n' = 10 \times n + d}{t \ \triangleright_n \ d \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex_dd}$$

$$\frac{t \cdot \mathsf{num}[n] \ \triangleright \ \epsilon \ \mathsf{lex_to} \ t'}{t \ \triangleright_n \ \epsilon \ \mathsf{lex_to} \ t'}\ \mathbf{lex_d\epsilon} \qquad \frac{t \cdot \mathsf{num}[n] \ \triangleright \ s \ \mathsf{lex_to} \ t'}{t \ \triangleright_n \ _ \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex_d_}$$

$$\frac{t \cdot \mathsf{num}[n] \ \triangleright \ \texttt{+} \cdot s \ \mathsf{lex_to} \ t'}{t \ \triangleright_n \ \texttt{+} \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex_d+} \qquad \frac{t \cdot \mathsf{num}[n] \ \triangleright \ \texttt{*} \cdot s \ \mathsf{lex_to} \ t'}{t \ \triangleright_n \ \texttt{*} \cdot s \ \mathsf{lex_to} \ t'}\ \mathbf{lex_d+}$$

The one interesting rule here is **lex_dd**, which recognizes an additional digit of a numeric string. Defining rules for the judgment in the right premise is rather simple but quite tedious. All the other rules simply call the main judgment after adding the token $\mathsf{num}[n]$ to the token stream.

Exercise

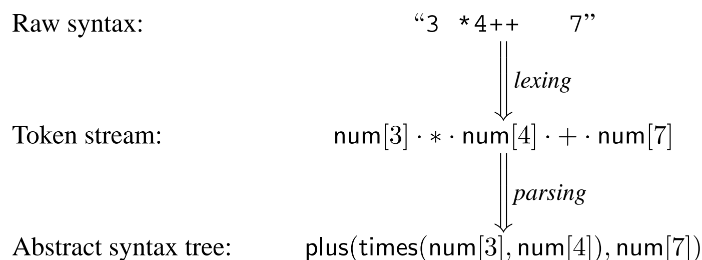
Define a syntax for regular expressions r and give the rules for a judgment s matches r which is derivable only when concrete string s matches regular expression r. It is easy to define this judgment non-deterministically by using string concatenation to split strings at appropriate places. You may want to model a more efficient version after some of the material you saw in 15-212 (a more general judgment is needed).

Assume you are given token forms $t \in T$ (for example all the tokens that can appear in a Java program) and rules implementing a judgment $s \rhd_r t$ that builds the token t out of the string s which matches regular expression r (assume you already know that s matches r; in the example above, one such judgment would build the token $\mathsf{num}[n]$ given a string of digits s and the regular expression $(\mathsf{0}|\ldots|\mathsf{9})^+$ and another would build $+$ when it sees++), generalize the above judgment $t \rhd s$ lex_to t' to produce a stream of these generic tokens.

Parsing

Moving from raw strings to token streams hides a substantial amount of irrelevant details for the purpose of examining a program as a mathematical object. Parsing exposes the structure of this flat sequence of tokens as defined by the grammar of the language. In particular, it turns tokens that are meant to connect other tokens (such as $*$ in $\mathsf{num}[3]\cdot*\cdot\mathsf{num}[4]$) into operators applied to arguments (here $\mathsf{times}(\mathsf{num}[3], \mathsf{num}[4])$). What we obtain as a result of parsing is an *abstract syntax tree*, often simply called a *term*. Programs written in this way are said to be in *abstract syntax*, as opposed to the *concrete syntax* of what is entered in a computer.

Consider the following example, starting from raw strings:



(the raw string has been compacted for clarity, and the trailing ϵ of the token stream has been dropped.)

The abstract syntax is defined inductively in a way that is coherent with the interpretation of a program or expression (recall that raw strings are also defined inductively,

but this definition has nothing to do with the higher-level meaning of a string, rather with the sequencing of characters). This inductive definition comes directly from the grammar for its language. For example, the grammar for (tokenized) expressions given earlier can be automatically transformed into the following inference rules for the judgment t exp, which interprets a token stream t as an expression:

$$\frac{}{\mathsf{num}[n]\ \mathsf{exp}}\ \mathsf{num}[\mathbf{n}] \qquad \frac{t_1\ \mathsf{exp} \quad t_2\ \mathsf{exp}}{t_1\hat{\ } + \hat{\ } t_2\ \mathsf{exp}}\ \mathsf{plus} \qquad \frac{t_1\ \mathsf{exp} \quad t_2\ \mathsf{exp}}{t_1\hat{\ } * \hat{\ } t_2\ \mathsf{exp}}\ \mathsf{times}$$

(here, we have taken some liberties: a fully formal definition would rely on the string splitting judgment s as $s_1\hat{\ }s_2$ defined earlier, adding one premise to the two rightmost rules.)

Consider the following derivation for the judgment

$$\mathsf{num}[3] \cdot * \cdot \mathsf{num}[4] \cdot + \cdot \mathsf{num}[7]\ \mathsf{exp}$$

involving the above example:

$$\frac{\dfrac{\dfrac{}{\mathsf{num}[3]\ \mathsf{exp}}\ \mathsf{num}[\mathbf{3}] \quad \dfrac{}{\mathsf{num}[4]\ \mathsf{exp}}\ \mathsf{num}[\mathbf{4}]}{\mathsf{num}[3] \cdot * \cdot \mathsf{num}[4]\ \mathsf{exp}}\ \mathsf{times} \quad \dfrac{}{\mathsf{num}[7]\ \mathsf{exp}}\ \mathsf{num}[\mathbf{7}]}{\mathsf{num}[3] \cdot * \cdot \mathsf{num}[4] \cdot + \cdot \mathsf{num}[7]\ \mathsf{exp}}\ \mathsf{plus}$$

This object is a parse tree for this expression, given as a derivation rather than in the usual way.

Now, the abstract syntax for this expression can be read off directly from this derivation by simply taking the names of the rules as operators. Indeed, taking the terms corresponding to the premises of a rule as the argument of this rule's name viewed as an operator, we obtain:

$$\mathsf{plus}(\mathsf{times}(\mathsf{num}[3], \mathsf{num}[4]), \mathsf{num}[7])$$

Therefore, the abstract syntax is just a simplified way of writing the parse tree for a (token) string. Here, we have justified it in terms of derivation. This technique is completely general, and scales to more complicated forms of abstract syntax.

This yields the following grammar for abstract expressions:

Abstract Expressions: $\quad E \ ::= \ \mathsf{num}[n] \ | \ \mathsf{plus}(E, E) \ | \ \mathsf{times}(E, E)$

We will generally omit the tag num.

This techniques is even more general than this: we can read any derivation as a term (although we often need a little bit more than just the rule names). This is the way proofs are entered into a theorem prover.

Harper's book (chapter 5) mentions an alternative way of building abstract syntax trees: rather than reading them off a parse tree in derivational form, they can be given an independent definition, and an additional judgment (parsing) is defined to relate strings (raw or tokenized) and abstract syntax trees. The net effect is the same, as this construction essentially defines the abstract syntax tree to be isomorphic to the derivation. This is another general approach — but sometimes error prone — to introducing terms to describe derivations.

Ambiguity

Of course, the above grammar for expressions is ambiguous, and the shown derivation tree is not the only one for our example (the other one takes $*$ to be the main operator, and therefore would build the term $\mathsf{times}(\mathsf{num}[3], \mathsf{plus}(\mathsf{num}[4], \mathsf{num}[7])))$. The ambiguity here derives from the fact that the string concatenation judgment $s \ \mathsf{as}\ s_1\hat{\ }s_2$, implicitly used in rules **plus** and **times**, is non-deterministic when used for splitting s into s_1 and s_2.

There are standard techniques for writing a grammar that ensure that it is non-ambiguous. Then, there is at most one parse tree (i.e., derivation in the sense above) in correspondence with any string in the language. See Harper's book (chapter 5) for a discussion on this.

Making a grammar non-ambiguous often involves introducing new non-terminals (which for us means adding judgments) and maybe parentheses. The construction given above clearly works also in this case, but doing so often carries over disambiguating machinery to the abstract syntax, which is not directly useful since the abstract syntax is not ambiguous (by definition). For this reason, the abstract syntax is generally based on a very simple (but usually ambiguous) grammar for a language.