Part II

Levels of Syntax

Chapter 5

Basic Syntactic Objects

We will make use of two sorts of objects for representing syntax, *strings* of characters, and *abstract syntax trees*. Strings provide a convenient representation for reading and entering programs, but are all but useless for manipulating programs as objects of study. Abstract syntax trees provide a representation of programs that exposes their hierarchical structure.

5.1 Symbols

We shall have use for a variety of *symbols*, which will serve in a variety of roles as characters, variable names, names of fields, and so forth. Symbols are sometimes called *names*, or *atoms*, or *identifiers*, according to custom in particular circumstances. Symbols are to be thought of as atoms with no structure other than their identity. We write $x \textsf{ sym}$ to assert that x is a symbol, and we assume that there are infinitely many symbols at our disposal. The judgement $x \# y$, where $x \textsf{ sym}$ and $y \textsf{ sym}$, states that x and y are distinct symbols.

We will make use of a variety of classes of symbols throughout the development. We generally assume that any two classes of symbols under consideration are disjoint from one another, so that there can be no confusion among them.

5.2 Strings Over An Alphabet

An *alphabet* is a (finite or infinite) collection of symbols, called *characters*. We write $c \textsf{ char}$ to indicate that c is a character, and let Σ stand for a finite set of such judgements, which is sometimes called an *alphabet*. The judgement $\Sigma \vdash s\ \mathsf{str}$, defining the strings over the alphabet Σ, is inductively defined by the following rules:

$$\frac{}{\Sigma \vdash \epsilon\ \mathsf{str}} \tag{5.1a}$$

$$\frac{\Sigma \vdash c\ \mathsf{char} \quad \Sigma \vdash s\ \mathsf{str}}{\Sigma \vdash c \cdot s\ \mathsf{str}} \tag{5.1b}$$

Thus a string is essentially a list of characters, with the null string being the empty list. We often suppress explicit mention of Σ when it is clear from context.

When specialized to Rules (5.1), the principle of rule induction states that to show $s\ P$ holds whenever $s\ \mathsf{str}$, it is enough to show

1. $\epsilon\ P$, and

2. if $s\ P$ and $c\ \mathsf{char}$, then $c \cdot s\ P$.

This is sometimes called the principle of *string induction*. It is essentially equivalent to induction over the length of a string, except that there is no need to define the length of a string in order to use it.

The following rules constitute an inductive definition of the judgement $s_1\,\hat{}\,s_2 = s\ \mathsf{str}$, stating that s is the result of concatenating the strings s_1 and s_2.

$$\frac{}{\epsilon\,\hat{}\,s = s\ \mathsf{str}} \tag{5.2a}$$

$$\frac{s_1\,\hat{}\,s_2 = s\ \mathsf{str}}{(c \cdot s_1)\,\hat{}\,s_2 = c \cdot s\ \mathsf{str}} \tag{5.2b}$$

It is easy to prove by string induction on the first argument that this judgement has mode $(\forall, \forall, \exists!)$. Thus, it determines a total function of its first two arguments.

Strings are usually written as juxtapositions of characters, writing just $abcd$ for the four-letter string $a \cdot (b \cdot (c \cdot (d \cdot \epsilon)))$, for example. Concatentation is also written as juxtaposition, and individual characters are often identified with the corresponding unit-length string. This means that $abcd$ can be thought of in many ways, for example as the concatenations $ab\,cd$, $a\,bcd$, or $abc\,d$, or even $\epsilon\,abcd$ or $abcd\,\epsilon$, as may be convenient in a given situation.

5.3 Abtract Syntax Trees

An *abstract syntax tree*, or *ast* for short, is an ordered tree in which certain symbols, called *operators*, label the nodes. A *signature*, Ω, is a finite set of judgements $\mathsf{ar}(o) = k$, where o is a symbol and $k \geq 0$, such that if $\Omega \vdash \mathsf{ar}(o) = k$ and $\Omega \vdash \mathsf{ar}(o) = k'$, then $k = k'$.

The class of abstract syntax trees over a signature, Ω, is inductively defined as follows.

$$\frac{\Omega \vdash \mathsf{ar}(o) = k \quad a_1\ \mathsf{ast} \quad \ldots \quad a_k\ \mathsf{ast}}{o(a_1, \ldots, a_k)\ \mathsf{ast}} \tag{5.3a}$$

The base case of this inductive definition is an operator of arity zero, in which case Rule (5.3a) has no premises. We often write just o, rather than $o()$, when this is the case.

We often write $\vdash_\Omega a\ \mathsf{ast}$, instead of $\Omega \vdash a\ \mathsf{ast}$, to state that a is an abstract syntax tree over signature Ω. In effect the signature plays the role of a rule set, since each arity assignment in the signature gives rise to a rule of the form (5.3a).

5.3.1 Structural Induction

The principle of *structural induction* is the specialization of the principle of rule induction to the rules defining ast's over a signature. To show that $\mathcal{P}(a\ \mathsf{ast})$, it is enough to show that $\mathcal{P}$ is closed under Rules (5.3). That is, if $\Omega \vdash \mathsf{ar}(o) = k$, then we are to show that

$$\textit{if } \mathcal{P}(a_1\ \mathsf{ast}),\ \ldots,\ \mathcal{P}(a_k\ \mathsf{ast}),\ \textit{then } \mathcal{P}(o(a_1, \ldots, a_k)\ \mathsf{ast}).$$

When n is zero, this reduces to showing that $\mathcal{P}(o())$.

For example, we consider the following inductive definition of the height of an abstract syntax tree:

$$\frac{\mathsf{hgt}(a_1) = n_1 \quad \ldots \quad \mathsf{hgt}(a_k) = n_k \quad \mathsf{max}(n_1, \ldots, n_k) = n \quad \mathsf{ar}(o) = k}{\mathsf{hgt}(o(a_1, \ldots, a_k)) = \mathtt{succ}(n)} \tag{5.4a}$$

We may prove by structural induction that this judgement has mode $(\forall, \exists!)$, which is to say that every ast has a unique height. For an operator o of arity k, we may assume by induction that, for each $1 \leq i \leq k$, there is a unique n_i such that $\mathsf{hgt}(a_i) = n_i$. We may show separately that the maximum, n, of these is uniquely determined, and hence that the overall height, $\mathtt{succ}(n)$, is also uniquely determined.

5.3.2 Variables and Substitution

The hypothetical judgement

$$x_1 \ \mathsf{ast}, \ldots, x_k \ \mathsf{ast} \vdash_\Omega a \ \mathsf{ast},$$

where the $x_1, \ldots, x_k$ are pairwise distinct symbols, states that a is an ast formed from the operators in Ω whose free variables are among $x_1, \ldots, x_k$.

Judgements such as the definition of the height of an ast are extended to account for variables by the use of hypotheses. For example, the judgement

$$\mathsf{hgt}(x_1) = 0, \ldots, \mathsf{hgt}(x_k) = 0 \vdash \mathsf{hgt}(a) = n$$

expresses that the ast a has height n, given that the heights of its variables are taken to be zero. There is nothing special about zero here; we could just as well assume that each variable has whatever height we like.

Lemma 5.1. *If* $x_1 \ \mathsf{ast}, \ldots, x_k \ \mathsf{ast} \vdash_\Omega a \ \mathsf{ast}$, *then for every* $n_1, \ldots, n_k$ *there exists a unique* $n \geq 0$ *such that* $\mathsf{hgt}(x_1) = n_1, \ldots, \mathsf{hgt}(x_k) = n_k \vdash \mathsf{hgt}(a) = n$.

Proof. By structural induction over the ast a. Bearing in mind the meaning of the derivability hypothetical judgement as enriching the rules with new axioms, the proof reduces to showing these two facts:

1. $\mathsf{hgt}(x_1) = n_1, \ldots, \mathsf{hgt}(x_k) = n_k \vdash \mathsf{hgt}(x_i) = n$ for each $1 \leq i \leq k$ and some $n \geq 0$ (namely, n_i). This follows immediately from the definition of derivability.

2. If $\Omega \vdash \mathsf{ar}(o) = k$, and if $\mathsf{hgt}(a_1) = n_1$, …, $\mathsf{hgt}(a_k) = n_k$ for uniquely determined $n_1, \ldots, n_k$, then $\mathsf{hgt}(o(a_1, \ldots, a_k)) = n$ for a unique n. This follows immediately from Rule (5.4a).

□

Substitution is defined for abstract syntax trees over a signature Ω by the following rules (one for each operator declared in Ω):

$$\frac{\Omega \vdash \mathsf{ar}(o) = k \quad [a/x]b_1 = c_1 \quad \ldots \quad [a/x]b_k = c_k}{[a/x]o(b_1, \ldots, b_k) = o(c_1, \ldots, c_k)} \tag{5.5a}$$

The judgement $[a/x]b = c$ is of interest only in the presence of assumptions governing the variables that may occur in b. The hypothetical judgement

$$[a/x]x_1 = x_1, \ldots, [a/x]x_k = x_k, [a/x]x = a \vdash [a/x]b = c$$

states that the result of substituting a for x in b is c, assuming that $[a/x]x = a$ and that $[a/x]x_i = x_i$ for all other variables $x_1, \ldots, x_k$.

Lemma 5.2. *If* x_1 ast, $\ldots$, x_k ast, x ast $\vdash b$ ast *and* x_1 ast, $\ldots$, x_k ast $\vdash a$ ast, *then there exists a unique* c *such that*

$$[a/x]x_1 = x_1, \ldots, [a/x]x_k = x_k, [a/x]x = a \vdash [a/x]b = c.$$

Proof. Similar to that of Lemma 5.1 on the facing page. □

In view of Lemma 5.2 we usually write $[a/x]b$ for the unique c such that $[a/x]b = c$, under the stated assumptions governing the variables in b. Simultaneous substitution, written $[a_1, \ldots, a_k/x_1, \ldots, x_k]b$, is defined similarly.

The definition of the height of an ast is stable under substitution.

Lemma 5.3. *If*

$$\mathsf{hgt}(x_1) = n_1, \ldots, \mathsf{hgt}(x_k) = n_k \vdash \mathsf{hgt}(a) = n,$$

and $\mathsf{hgt}(a_1) = n_1$, $\ldots$, $\mathsf{hgt}(a_k) = n_k$, *then there exists a unique* p *such that* $\mathsf{hgt}([a_1, \ldots, a_k/x_1, \ldots, x_k]a) = p$.

Lemma 5.3 may be succinctly stated using a generic judgement over abstract syntax trees (with substitution as just defined) as follows:

$$\{ x_1, \ldots, x_k \} \mid \mathsf{hgt}(x_1) = n_1, \ldots, \mathsf{hgt}(x_k) = n_k \vdash \mathsf{hgt}(a) = n.$$

In practice we rely on typographical conventions to determine the set of parameters over which the judgement is generic. Under the convention that the x_i's are to be taken as parameters, the foregoing judgement may be abbreviated by writing just

$$\mathsf{hgt}(x_1) = n_1, \ldots, \mathsf{hgt}(x_k) = n_k \vdash \mathsf{hgt}(a) = n.$$

However, it is important to bear in mind when using this notation that the parameters may be renamed as discussed in Chapter 3.

5.4 Exercises

Chapter 6

Binding and Scope

Abstract syntax trees expose the hierarchical structure of syntax, dispensing with the details of how one might represent pieces of syntax on a page or a computer screen. *Abstract binding trees*, or *abt's*, enrich this representation with the concepts of *binding* and *scope*. In just about every language there is a means of associating a meaning to an identifier within a specified range of significance (perhaps the whole program, often limited regions of it). Examples include definitions, in which we introduce a name for a program phrase, or parameters to functions, in which we introduce a name for the argument to the function within its body.

Abstract binding trees enrich abstract syntax trees with a means of introducing a *fresh*, or *new*, name for use within a specified scope. Uses of the fresh name within that scope are references to the binding site. As such the particular choice of name is significant only insofar as it does not conflict with any other name currently in scope; this is the essence of what it means for the name to be "new" or "fresh."

In this chapter we introduce the concept of an abstract binding tree, including the relation of α-equivalence, which expresses the irrelevance of the choice of bound names, and the operation of *capture-avoiding substitution*, which ensures that names are not confused by substitution. While intuitively clear, the precise formalization of these concepts requires some care; experience has shown that it is surprisingly easy to get them wrong.

All of the programming languages that we shall study are represented as abstract binding trees. Consequently, we will re-use the machinery developed in this chapter many times, avoiding considerable redundancy and consolidating the effort required to make precise the notions of binding and scope.

6.1 Abstract Binding Trees

The concepts of binding and scope are formalized by the concept of an *abstract binding tree*, or *abt*. An abt is an ast in which we distinguish a name-indexed family of operators, called *abstractors*. An abstractor has the form $x\,.\,a$; it *binds* the name, x, for use in the abt, a, which is called the *scope* of the binding. The bound name x is meaningful only within a, and is, in a sense to be made precise shortly, treated as distinct from any other names that may be currently in scope.

As with abstract syntax trees, the well-formed abstract binding trees are determined by a *signature* that specifies the *arity* of each of a finite collection of operators. For ast's the arity specified only the number of arguments for each operator, but for abt's we must also specify the number of names that are bound by each operator. Thus an arity is a finite sequence $(n_1, \ldots, n_k)$ of natural numbers, with k specifying the number of arguments, and each n_i specifying the *valence*, or number of bound names, in the ith argument. The arity $(0, 0, \ldots, 0)$, of length k specifies an operator with k arguments that binds no variables in any argument; it is therefore the analogue of the arity k for an operator over abstract syntax trees.

A signature, Ω, consists of a finite set of judgements of the form $\mathsf{ar}(o) = (n_1, \ldots, n_k)$ such that no operator occurs in more than one such judgement. The well-formed abt's over a signature Ω are specified by a hypothetical judgement of the form

$$x_1\ \mathsf{abt}^0, \ldots, x_k\ \mathsf{abt}^0 \vdash a\ \mathsf{abt}^n$$

stating that a is an abt of *valence n*, with *free variables* $x_1, \ldots, x_k$. We sometimes write just $a\ \mathsf{abt}$ as short-hand for $a\ \mathsf{abt}^0$.

We use the meta-variable $\mathcal{A}$ to range over finite sets of assumptions of the form $x\ \mathsf{abt}^0$, where x is a parameter. We write $x \# \mathcal{A}$ to mean that there is no assumption of the form $x\ \mathsf{abt}^0$ in $\mathcal{A}$.

The rules defining the well-formed abt's over a given signature are as follows:

$$\frac{}{\mathcal{A}, x\ \mathsf{abt}^0 \vdash x\ \mathsf{abt}^0} \tag{6.1a}$$

$$\frac{\mathsf{ar}(o) = (n_1, \ldots, n_k) \quad \mathcal{A} \vdash a_1\ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{A} \vdash a_k\ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash o(a_1, \ldots, a_k)\ \mathsf{abt}^0} \tag{6.1b}$$

$$\frac{\mathcal{A}, x'\ \mathsf{abt}^0 \vdash [x' \leftrightarrow x]\, a\ \mathsf{abt}^n \quad (x' \# \mathcal{A})}{\mathcal{A} \vdash x\,.\,a\ \mathsf{abt}^{n+1}} \tag{6.1c}$$

Rule (6.1c) specifies that an abstractor, $x\,.\,a$, is well-formed relative to $\mathcal{A}$, provided that its body, a, is well-formed for some variable $x' \mathrel{\#} \mathcal{A}$ replacing the bound variable, x, in a. The replacement of x by x' ensures that the formation of $x\,.\,a$ does not depend on whether x is already an active parameter.

6.1.1 Structural Induction With Binding and Scope

The principle of structural induction for abstract syntax trees extends to abstract binding trees. To show that $\mathcal{P}(\mathcal{A} \vdash a\ \mathsf{abt}^n)$ whenever $\mathcal{A} \vdash a\ \mathsf{abt}^n$, it suffices to show that $\mathcal{P}$ is closed under Rules (6.1). Specifically, we must show:

1. $\mathcal{P}(\mathcal{A}, x\ \mathsf{abt}^0 \vdash x\ \mathsf{abt}^0)$.
2. If o has arity $(m_1, \dots, m_k)$ and $\mathcal{P}(\mathcal{A} \vdash a_1\ \mathsf{abt}^{m_1}), \dots, \mathcal{P}(\mathcal{A} \vdash a_k\ \mathsf{abt}^{m_k})$, then $\mathcal{P}(\mathcal{A} \vdash o(a_1, \dots, a_k)\ \mathsf{abt}^0)$.
3. If $\mathcal{P}(\mathcal{A}, x'\ \mathsf{abt}^0 \vdash [x' \leftrightarrow x]\, a\ \mathsf{abt}^n)$ for every $x' \mathrel{\#} \mathcal{A}$, then $\mathcal{P}(\mathcal{A} \vdash x\,.\,a\ \mathsf{abt}^{n+1})$.

The condition on abstractors ensures that the name of a bound variable does not matter, and permits us to consider that it is chosen to be any fresh variable of our choosing.[1]

As an example let us define the size, s, of an abt, a, of valence n by a judgement of the form $\mathsf{sz}(a\ \mathsf{abt}^n) = s$. More generally, we define the hypothetical judgement

$$\mathsf{sz}(x_1\ \mathsf{abt}^0) = 1, \dots, \mathsf{sz}(x_k\ \mathsf{abt}^0) = 1 \vdash \mathsf{sz}(a\ \mathsf{abt}^n) = s,$$

with implied parameters $x_1, \dots, x_k$, by the following rules:

$$\frac{}{\mathcal{S}, \mathsf{sz}(x\ \mathsf{abt}^0) = 1 \vdash \mathsf{sz}(x\ \mathsf{abt}^0) = 1} \tag{6.2a}$$

$$\frac{\mathcal{S} \vdash \mathsf{sz}(a_1\ \mathsf{abt}^{n_1}) = s_1 \quad \dots \quad \mathcal{S} \vdash \mathsf{sz}(a_m\ \mathsf{abt}^{n_m}) = s_m \quad s = s_1 + \cdots + s_m + 1}{\mathcal{S} \vdash \mathsf{sz}(o(a_1, \dots, a_m)\ \mathsf{abt}^0) = s} \tag{6.2b}$$

[1]There is a sticky technical issue lurking here regarding the discrepancy between Rule (6.1c), which is stated for *some* choice of fresh variable x', and the induction principle, which is stated for *every* choice of fresh variable x'. This may be justified by re-stating Rule (6.1c) to consider every choice of x', justifying the induction principle, and showing that Rule (6.1c) is admissible, allowing us to make a particular choice when exhibiting a derivation.

$$\frac{\mathcal{S}, \mathsf{sz}(x' \ \mathsf{abt}^0) = 1 \vdash \mathsf{sz}([x \leftrightarrow x']\, a \ \mathsf{abt}^n) = s}{\mathcal{S} \vdash \mathsf{sz}(x\,.\,a \ \mathsf{abt}^{n+1}) = s + 1} \tag{6.2c}$$

Thus, the size of an abt is defined inductively counting variables as unit size, and adding one for each operator and abstractor within the abt.

Theorem 6.1. *Every well-formed abt has a unique size. If $x_1 \ \mathsf{abt}^0, \ldots, x_k \ \mathsf{abt}^0 \vdash a \ \mathsf{abt}^n$, then there exists a unique s nat such that*

$$\mathsf{sz}(x_1 \ \mathsf{abt}^0) = 1, \ldots, \mathsf{sz}(x_k \ \mathsf{abt}^0) = 1 \vdash \mathsf{sz}(a \ \mathsf{abt}^n) = s.$$

Proof. By structural induction on the derivation of the premise. Note that the size of an abt is not sensitive to the choice of parameters, since all parameters are assigned unit size. It is straightforward to show that this property is closed under the given rules and to show that the size is uniquely determined for well-formed abt's. □

6.1.2 Renaming of Bound Names

Two abt's are said to be *α-equivalent* iff they differ at most in the choice of bound variable names. It is inductively defined by the following rules:

$$\frac{}{\mathcal{A}, x \ \mathsf{abt}^0 \vdash x =_\alpha x \ \mathsf{abt}^0} \tag{6.3a}$$

$$\frac{\mathcal{A} \vdash a_1 =_\alpha b_1 \ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{A} \vdash a_k =_\alpha b_k \ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash o(a_1, \ldots, a_k) =_\alpha o(b_1, \ldots, b_k) \ \mathsf{abt}^0} \tag{6.3b}$$

$$\frac{\mathcal{A}, z \ \mathsf{abt}^0 \vdash [z \leftrightarrow x]\, a =_\alpha [z \leftrightarrow y]\, b \ \mathsf{abt}^n}{\mathcal{A} \vdash x\,.\,a =_\alpha y\,.\,b \ \mathsf{abt}^{n+1}} \tag{6.3c}$$

In Rule (6.3c) we tacitly assume that the parameter z is chosen apart from those in $\mathcal{A}$.

We write $\mathcal{A} \vdash a =_\alpha b$ for $\mathcal{A} \vdash a =_\alpha b \ \mathsf{abt}^n$ for some n. Further, we sometimes write just $a =_\alpha b$ to mean $\mathcal{A} \vdash a =_\alpha b$ when the appropriate $\mathcal{A}$ is clear from context.

Lemma 6.2. *The following instance of α-equivalence, called* α-conversion, *is derivable:*

$$\mathcal{A} \vdash x\,.\,a =_\alpha y\,.\,[x \leftrightarrow y]\, a \ \mathsf{abt}^{n+1} \qquad (y \mathrel{\#} \mathcal{A}).$$

Theorem 6.3. *α-equivalence is reflexive, symmetric, and transitive.*

Proof. Reflexivity and symmetry are immediately obvious from the form of the definition. Transitivity is proved by a simultaneous induction on the heights of the derivations of $\mathcal{A} \vdash a =_\alpha b \ \mathsf{abt}^n$ and $\mathcal{A} \vdash b =_\alpha c \ \mathsf{abt}^n$. The most interesting case is when both derivations end with Rule (6.3c). We have $a = x\,.\,a'$, $b = y\,.\,b'$, $c = z\,.\,c'$, and $n = m + 1$ for some m. Moreover, $\mathcal{A}, u \ \mathsf{abt}^0 \vdash [u \leftrightarrow x]\, a' =_\alpha [u \leftrightarrow y]\, b' \ \mathsf{abt}^m$, and $\mathcal{A}, v \ \mathsf{abt}^0 \vdash [v \leftrightarrow y]\, b' =_\alpha [v \leftrightarrow z]\, c' \ \mathsf{abt}^m$, for every $u, v \ \# \ \mathcal{A}$. Let $w \ \# \ \mathcal{A}$ be an arbitrary name. By choosing u and v to be w, we obtain the desired result by an application of the inductive hypothesis. □

6.1.3 Capture-Avoiding Substitution

Substitution is the process of replacing all occurrences (if any) of a free name in an abt by another abt in such a way that the scopes of names are properly respected. The judgment $\mathcal{A} \vdash [a/x]b = c \ \mathsf{abt}^n$ is inductively defined by the following rules:

$$\frac{}{\mathcal{A} \vdash [a/x]x = a \ \mathsf{abt}^0} \tag{6.4a}$$

$$\frac{x \ \# \ y}{\mathcal{A} \vdash [a/x]y = y \ \mathsf{abt}^0} \tag{6.4b}$$

$$\frac{\mathcal{A} \vdash [a/x]b_1 = c_1 \ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{A} \vdash [a/x]b_k = c_k \ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash [a/x]o(b_1, \ldots, b_k) = o(c_1, \ldots, c_k) \ \mathsf{abt}^0} \tag{6.4c}$$

$$\frac{\mathcal{A}, y' \ \mathsf{abt}^0 \vdash [a/x]([y' \leftrightarrow y]\, b) = b' \ \mathsf{abt}^n \quad y' \ \# \ \mathcal{A} \quad y' \neq x}{\mathcal{A} \vdash [a/x]y\,.\,b = y'\,.\,b' \ \mathsf{abt}^n} \tag{6.4d}$$

In Rule (6.4d) the requirement that $y' \ \# \ \mathcal{A}$ ensures that $y' \ \# \ a$, and the requirement that $y' \neq x$ ensures that we do not confuse y' with x. Since the bound name, y, of the abstractor might well occur within $\mathcal{A}$, it may also occur in a. This necessitates that y be renamed to a fresh name y' before substituting a into the body of the abstractor. The potential confusion of an occurrence of y within a with the bound variable of the abstractor is called *capture*, and for this reason substitution as defined here is called *capture-avoiding substitution*.

The penalty for avoiding capture during substitution is that the result of performing a substitution is determined only up to α-equivalence. Observe that in the conclusion of Rule (6.4d), we have $y\,.\,[y \leftrightarrow y']\, b' =_\alpha y'\,.\,b'$, provided that $y \ \# \ \mathcal{A}$, by Lemma 6.2 on the preceding page. If, on the contrary, y occurs within $\mathcal{A}$, then the equivalence does not apply, and, as a consequence, we cannot preserve the bound name after substitution.

Theorem 6.4. *If $\mathcal{A} \vdash a$ abt^0 and $\mathcal{A}, x$ $\mathsf{abt}^0 \vdash b$ abt^n, then there exists $\mathcal{A} \vdash c$ abt^n such that $\mathcal{A} \vdash [a/x]b = c$ abt^n. If $\mathcal{A} \vdash [a/x]b = c$ abt^n and $\mathcal{A} \vdash [a/x]b = c'$ abt^n, then $\mathcal{A} \vdash c =_\alpha c'$ abt^n.*

Proof. The first part is proved by rule induction on $\mathcal{A}, x$ $\mathsf{abt}^0 \vdash b$ abt^n, in each case constructing the required derivation of the substitution judgement. The second part is proved by simultaneous rule induction on the two premises, deriving the desired equivalence in each case. □

Even though the result is not uniquely determined, we abuse notation and write $[a/x]b$ for any c such that $[a/x]b = c$, with the understanding that c is determined only up to choice of bound names. To ensure that this convention is sensible, we will ensure that all judgements on abt's are defined so as to respect α-equivalence—in particular, substitution itself enjoys this property.

Theorem 6.5. *If $\mathcal{A} \vdash a =_\alpha a'$ abt^0, $\mathcal{A}, x$ $\mathsf{abt}^0 \vdash b =_\alpha b'$ abt^n, $\mathcal{A} \vdash [a/x]b = c$ abt^n and $\mathcal{A} \vdash [a'/x]b' = c'$ abt^n, then $\mathcal{A} \vdash c =_\alpha c'$ abt^n.*

Proof. By rule induction on $\mathcal{A}, x$ $\mathsf{abt}^0 \vdash b =_\alpha b'$ abt^n. □

6.2 Generic Judgements Over ABT's

Generic judgements over abstract binding trees are used to state the validity of a judgement under all substitutions for its parameters. For example, the generic judgement

$$\{ x_1, \ldots, x_n \} \mid \mathsf{sz}(x_1\ \mathsf{abt}^0) = s_1, \ldots, \mathsf{sz}(x_k\ \mathsf{abt}^0) = s_k \vdash \mathsf{sz}(a\ \mathsf{abt}^n) = s \quad (6.5)$$

states that the hypothetical judgement

$$\mathsf{sz}(a_1\ \mathsf{abt}^0) = s_1, \ldots, \mathsf{sz}(a_k\ \mathsf{abt}^0) = s_k \vdash \mathsf{sz}([a_1, \ldots, a_k / x_1, \ldots, x_k]a\ \mathsf{abt}^n) = s$$

holds for every substitution of abt's a_i for x_i for each $1 \le i \le k$. Consequently, by the transitivity of the hypothetical judgement, if $\mathsf{sz}(a_i\ \mathsf{abt}^0) = s_i$ for each $1 \le i \le n$, then $\mathsf{sz}([a_1, \ldots, a_k / x_1, \ldots, x_k]a\ \mathsf{abt}^n) = s$.

In most cases judgements about abt's are intended to hold for all substitution instances with respect to the variables involved. In such situations it is convenient to use a generic inductive definition. For example, For example, here is a generic inductive definition of the size of an abt:

$$\frac{}{\mathcal{X} \mid \mathcal{S}, \mathsf{sz}(x\ \mathsf{abt}^0) = s \vdash \mathsf{sz}(x\ \mathsf{abt}^0) = s} \quad (6.6a)$$

$$\frac{s = s_1 + \cdots + s_m + 1 \quad \mathcal{X} \mid \mathcal{S} \vdash \mathsf{sz}(a_1 \ \mathsf{abt}^{n_1}) = s_1 \quad \ldots \quad \mathcal{X} \mid \mathcal{S} \vdash \mathsf{sz}(a_m \ \mathsf{abt}^{n_m}) = s_m}{\mathcal{X} \mid \mathcal{S} \vdash \mathsf{sz}(o(a_1, \ldots, a_m) \ \mathsf{abt}^0) = s} \tag{6.6b}$$

$$\frac{\mathcal{X}, x \mid \mathcal{S}, \mathsf{sz}(x \ \mathsf{abt}^0) = 1 \vdash \mathsf{sz}(a \ \mathsf{abt}^n) = s}{\mathcal{X} \mid \mathcal{S} \vdash \mathsf{sz}(x\,.\,a \ \mathsf{abt}^{n+1}) = s + 1} \tag{6.6c}$$

It is implicit in Rule (6.6c) that the variable x is chosen so that it does not occur in $\mathcal{X}$. This requirement may always be met, provided that the following rule of α-equivalence is tacitly included in any generic inductive definition:

$$\frac{\mathcal{X} \mid \mathcal{S} \vdash \mathsf{sz}(a' \ \mathsf{abt}^n) = s \quad a =_\alpha a'}{\mathcal{X} \mid \mathcal{S} \vdash \mathsf{sz}(a \ \mathsf{abt}^n) = s}\ . \tag{6.7}$$

In short we tacitly identify abt's up to α-equivalence, and demand that all judgements respect this relation. (Besides affording a notational convenience in the phrasing of the rules, inclusion of this rule is also critical for ensuring that the structural rule of instantiation (Rule (3.8b)) is admissible.)

It is a common notational practice to drop explicit mention of the set of variables in a generic judgement when it is clear what this set should be. For example, we may write

$$\mathsf{sz}(x_1 \ \mathsf{abt}^0) = s_1, \ldots, \mathsf{sz}(x_k \ \mathsf{abt}^0) = s_k \vdash \mathsf{sz}(a \ \mathsf{abt}^n) = s,$$

for the generic judgement

$$\{\, x_1, \ldots, x_k \,\} \mid \mathsf{sz}(x_1 \ \mathsf{abt}^0) = s_1, \ldots, \mathsf{sz}(x_k \ \mathsf{abt}^0) = s_k \vdash \mathsf{sz}(a \ \mathsf{abt}^n) = s.$$

We rely on typographical conventions to make clear which are the parameters of the judgement in such situations.

6.3 Exercises

1. Show that the structural rule of weakening is *not* admissible for the hypothetical inductive definition of abstract binding trees (Rules (6.1)).

2. Suppose that `let` is an operator of arity $(0, 1)$ and that `plus` is an operator of arity $(0, 0)$. Determine whether or not each of the following

α-equivalences are valid.

$$\texttt{let}(x,x.x) =_\alpha \texttt{let}(x,y.y) \tag{6.8a}$$
$$\texttt{let}(y,x.x) =_\alpha \texttt{let}(y,y.y) \tag{6.8b}$$
$$\texttt{let}(x,x.x) =_\alpha \texttt{let}(y,y.y) \tag{6.8c}$$
$$\texttt{let}(x,x.\texttt{plus}(x,y)) =_\alpha \texttt{let}(x,z.\texttt{plus}(z,y)) \tag{6.8d}$$
$$\texttt{let}(x,x.\texttt{plus}(x,y)) =_\alpha \texttt{let}(x,y.\texttt{plus}(y,y)) \tag{6.8e}$$

3. Prove that apartness respects α-equivalence.

4. Prove that substitution respects α-equivalence.

Chapter 7

Concrete Syntax

The *concrete syntax* of a language is a means of representing expressions as strings that may be written on a page or entered using a keyboard. The concrete syntax usually is designed to enhance readability and to eliminate ambiguity. While there are good methods for eliminating ambiguity, improving readability is, to a large extent, a matter of taste.

In this chapter we introduce the main methods for specifying concrete syntax, using as an example an illustrative expression language, called $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, that supports elementary arithmetic on the natural numbers and simple computations on strings. In addition, $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ includes a construct for binding the value of an expression to a variable within a specified scope.

7.1 Lexical Structure

The first phase of syntactic processing is to convert from a character-based representation to a symbol-based representation of the input. This is called *lexical analysis*, or *lexing*. The main idea is to aggregate characters into symbols that serve as tokens for subsequent phases of analysis. For example, the numeral `467` is written as a sequence of three consecutive characters, one for each digit, but is regarded as a single token, namely the number 467. Similarly, an identifier such as `temp` comprises four letters, but is treated as a single symbol representing the entire word. Moreover, many character-based representations include empty "white space" (spaces, tabs, newlines, and, perhaps, comments) that are discarded by the lexical analyzer.[1]

[1]In some languages white space *is* significant, in which case it must be converted to symbolic form for subsequent processing.

The character representation of symbols is, in most cases, conveniently described using *regular expressions*. The lexical structure of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is specified as follows:

Item	itm	$::=$	$\mathsf{kwd} \mid \mathsf{id} \mid \mathsf{num} \mid \mathsf{lit} \mid \mathsf{spl}$
Keyword	kwd	$::=$	$\texttt{l}\cdot\texttt{e}\cdot\texttt{t}\cdot\epsilon \mid \texttt{b}\cdot\texttt{e}\cdot\epsilon \mid \texttt{i}\cdot\texttt{n}\cdot\epsilon$
Identifier	id	$::=$	$\mathsf{ltr}\,(\mathsf{ltr} \mid \mathsf{dig})^*$
Numeral	num	$::=$	$\mathsf{dig}\,\mathsf{dig}^*$
Literal	lit	$::=$	$\mathsf{qum}\,(\mathsf{ltr} \mid \mathsf{dig})^*\mathsf{qum}$
Special	spl	$::=$	$\texttt{+} \mid \texttt{*} \mid \texttt{\^{}} \mid \texttt{(} \mid \texttt{)} \mid \vert$
Letter	ltr	$::=$	$\texttt{a} \mid \texttt{b} \mid \ldots$
Digit	dig	$::=$	$\texttt{0} \mid \texttt{1} \mid \ldots$
Quote	qum	$::=$	$\texttt{"}$

A lexical item is either a keyword, an identifier, a numeral, a string literal, or a special symbol. There are three keywords, specified as sequences of characters, for emphasis. Identifiers start with a letter and may involve subsequent letters or digits. Numerals are non-empty sequences of digits. String literals are sequences of letters or digits surrounded by quotes. The special symbols, letters, digits, and quote marks are as enumerated. (Observe that we tacitly identify a character with the unit-length string consisting of that character.)

The job of the lexical analyzer is to translate character strings into token strings using the above definitions as a guide. An input string is scanned, ignoring white space, and translating lexical items into tokens, which are specified by the following rules:

$$\frac{s\ \mathsf{str}}{\texttt{ID[}s\texttt{]}\ \mathsf{tok}} \tag{7.1a}$$

$$\frac{n\ \mathsf{nat}}{\texttt{NUM[}n\texttt{]}\ \mathsf{tok}} \tag{7.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{LIT[}s\texttt{]}\ \mathsf{tok}} \tag{7.1c}$$

$$\frac{}{\texttt{LET}\ \mathsf{tok}} \tag{7.1d}$$

$$\frac{}{\texttt{BE}\ \mathsf{tok}} \tag{7.1e}$$

$$\frac{}{\texttt{IN}\ \mathsf{tok}} \tag{7.1f}$$

$$\frac{}{\texttt{ADD}\ \mathsf{tok}} \tag{7.1g}$$

$$\frac{}{\texttt{MUL}\ \mathsf{tok}} \tag{7.1h}$$

$$\frac{}{\texttt{CAT}\ \mathsf{tok}} \tag{7.1i}$$

$$\frac{}{\texttt{LP}\ \mathsf{tok}} \tag{7.1j}$$

$$\frac{}{\texttt{RP}\ \mathsf{tok}} \tag{7.1k}$$

$$\frac{}{\texttt{VB}\ \mathsf{tok}} \tag{7.1l}$$

Lexical analysis is inductively defined by the following judgement forms:

$s\ \mathsf{inp} \longleftrightarrow t\ \mathsf{tokstr}$	Scan input
$s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}$	Scan an item
$s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}$	Scan a keyword
$s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}$	Scan an identifier
$s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}$	Scan a number
$s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}$	Scan a symbol
$s\ \mathsf{lit} \longleftrightarrow t\ \mathsf{tok}$	Scan a string literal
$s\ \mathsf{whs}$	Skip white space

The definition of these forms, which follows, makes use of several auxiliary judgements corresponding to the classifications of characters in the lexical structure of the language. For example, s whs states that the string s consists only of "white space", and s lord states that s is either an alphabetic letter or a digit, and so forth.

$$\frac{}{\epsilon\ \mathsf{inp} \longleftrightarrow \epsilon\ \mathsf{tokstr}} \tag{7.2a}$$

$$\frac{s = s_1\,\hat{}\,s_2\,\hat{}\,s_3\ \mathsf{str} \quad s_1\ \mathsf{whs} \quad s_2\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok} \quad s_3\ \mathsf{inp} \longleftrightarrow ts\ \mathsf{tokstr}}{s\ \mathsf{inp} \longleftrightarrow t \cdot ts\ \mathsf{tokstr}} \tag{7.2b}$$

$$\frac{s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2c}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2d}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2e}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2f}$$

$$\frac{s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2g}$$

$$\frac{s = \texttt{l} \cdot \texttt{e} \cdot \texttt{t} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \texttt{LET}\ \mathsf{tok}} \tag{7.2h}$$

$$\frac{s = \texttt{b} \cdot \texttt{e} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \texttt{BE}\ \mathsf{tok}} \tag{7.2i}$$

$$\frac{s = \mathtt{i} \cdot \mathtt{n} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{kwd} \longleftrightarrow \mathtt{IN} \ \mathsf{tok}} \tag{7.2j}$$

$$\frac{s = s_1 \mathbin{\hat{}} s_2 \ \mathsf{str} \quad s_1 \ \mathsf{ltr} \quad s_2 \ \mathsf{lord}}{s \ \mathsf{id} \longleftrightarrow \mathtt{ID}[s] \ \mathsf{tok}} \tag{7.2k}$$

$$\frac{s = s_1 \mathbin{\hat{}} s_2 \ \mathsf{str} \quad s_1 \ \mathsf{dig} \quad s_2 \ \mathsf{dgs} \quad s \ \mathsf{num} \longleftrightarrow n \ \mathsf{nat}}{s \ \mathsf{num} \longleftrightarrow \mathtt{NUM}[n] \ \mathsf{tok}} \tag{7.2l}$$

$$\frac{s = s_1 \mathbin{\hat{}} s_2 \mathbin{\hat{}} s_3 \ \mathsf{str} \quad s_1 \ \mathsf{qum} \quad s_2 \ \mathsf{lord} \quad s_3 \ \mathsf{qum}}{s \ \mathsf{lit} \longleftrightarrow \mathtt{LIT}[s_2] \ \mathsf{tok}} \tag{7.2m}$$

$$\frac{s = \mathtt{+} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{ADD} \ \mathsf{tok}} \tag{7.2n}$$

$$\frac{s = \mathtt{*} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{MUL} \ \mathsf{tok}} \tag{7.2o}$$

$$\frac{s = \mathtt{\hat{}} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{CAT} \ \mathsf{tok}} \tag{7.2p}$$

$$\frac{s = \mathtt{(} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{LP} \ \mathsf{tok}} \tag{7.2q}$$

$$\frac{s = \mathtt{)} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{RP} \ \mathsf{tok}} \tag{7.2r}$$

$$\frac{s = \mathtt{|} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{VB} \ \mathsf{tok}} \tag{7.2s}$$

By convention Rule (7.2k) applies only if none of Rules (7.2h) to (7.2j) apply. Technically, Rule (7.2k) has implicit premises that rule out keywords as possible identifiers.

7.2 Context-Free Grammars

The standard method for defining concrete syntax is by giving a *context-free grammar* for the language. A grammar consists of three components:

1. The *tokens*, or *terminals*, over which the grammar is defined.

2. The *syntactic classes*, or *non-terminals*, which are disjoint from the terminals.

3. The *rules*, or *productions*, which have the form $A ::= \alpha$, where A is a non-terminal and α is a string of terminals and non-terminals.

Each syntactic class is a collection of token strings. The rules determine which strings belong to which syntactic classes.

When defining a grammar, we often abbreviate a set of productions,

$$\begin{gathered} A ::= \alpha_1 \\ \vdots \\ A ::= \alpha_n, \end{gathered}$$

each with the same left-hand side, by the *compound* production

$$A ::= \alpha_1 \mid \ldots \mid \alpha_n,$$

which specifies a set of alternatives for the syntactic class A.

A context-free grammar determines a simultaneous inductive definition of its syntactic classes. Specifically, we regard each non-terminal, A, as a judgement form, $s\ A$, over strings of terminals. To each production of the form

$$A ::= s_1\, A_1\, s_2\, \ldots\, s_n\, A_n\, s_{n+1} \tag{7.3}$$

we associate an inference rule

$$\frac{s_1'\ A_1 \quad \ldots \quad s_n'\ A_n}{s_1\, s_1'\, s_2\, \ldots\, s_n\, s_n'\, s_{n+1}\ A}\ . \tag{7.4}$$

The collection of all such rules constitutes an inductive definition of the syntactic classes of the grammar.

Recalling that juxtaposition of strings is short-hand for their concatenation, we may re-write the preceding rule as follows:

$$\frac{s_1'\ A_1 \quad \ldots \quad s_n'\ A_n \quad s = s_1\,\hat{}\,s_1'\,\hat{}\,s_2\,\hat{}\ldots s_n\,\hat{}\,s_n'\,\hat{}\,s_{n+1}}{s\ A}\ . \tag{7.5}$$

This formulation makes clear that $s\ A$ holds whenever s can be partitioned as described so that $s_i'\ A$ for each $1 \leq i \leq n$. Since string concatenation is not invertible, the decomposition is not unique, and so there may be many different ways in which the rule applies.

7.3 Grammatical Structure

The concrete syntax of $\mathcal{L}\{$`num str`$\}$ may be specified by a context-free grammar over the tokens defined in Section 7.1 on page 53. The grammar has only one syntactic class, exp, which is defined by the following compound production:

$$\begin{array}{lrcl}
\text{Expression} & \mathsf{exp} & ::= & \mathsf{num} \mid \mathsf{lit} \mid \mathsf{id} \mid \mathtt{LP}\,\mathsf{exp}\,\mathtt{RP} \mid \mathsf{exp}\,\mathtt{ADD}\,\mathsf{exp} \mid \\
 & & & \mathsf{exp}\,\mathtt{MUL}\,\mathsf{exp} \mid \mathsf{exp}\,\mathtt{CAT}\,\mathsf{exp} \mid \mathtt{VB}\,\mathsf{exp}\,\mathtt{VB} \mid \\
 & & & \mathtt{LET}\,\mathsf{id}\,\mathtt{BE}\,\mathsf{exp}\,\mathtt{IN}\,\mathsf{exp} \\
\text{Number} & \mathsf{num} & ::= & \mathtt{NUM[}n\mathtt{]} \quad (n\ \mathsf{nat}) \\
\text{String} & \mathsf{lit} & ::= & \mathtt{LIT[}s\mathtt{]} \quad (s\ \mathsf{str}) \\
\text{Identifier} & \mathsf{id} & ::= & \mathtt{ID[}s\mathtt{]} \quad (s\ \mathsf{str})
\end{array}$$

This grammar makes use of some standard notational conventions to improve readability: we identify a token with the corresponding unit-length string, and we use juxtaposition to denote string concatenation.

Applying the interpretation of a grammar as an inductive definition, we obtain the following rules:

$$\frac{s\ \mathsf{num}}{s\ \mathsf{exp}} \tag{7.6a}$$

$$\frac{s\ \mathsf{lit}}{s\ \mathsf{exp}} \tag{7.6b}$$

$$\frac{s\ \mathsf{id}}{s\ \mathsf{exp}} \tag{7.6c}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\,\mathtt{ADD}\,s_2\ \mathsf{exp}} \tag{7.6d}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\,\mathtt{MUL}\,s_2\ \mathsf{exp}} \tag{7.6e}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\,\mathtt{CAT}\,s_2\ \mathsf{exp}} \tag{7.6f}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{VB}\,s\,\mathtt{VB}\ \mathsf{exp}} \tag{7.6g}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{LP}\,s\,\mathtt{RP}\ \mathsf{exp}} \tag{7.6h}$$

$$\frac{s_1\ \mathsf{id} \quad s_2\ \mathsf{exp} \quad s_3\ \mathsf{exp}}{\mathtt{LET}\,s_1\,\mathtt{BE}\,s_2\,\mathtt{IN}\,s_3\ \mathsf{exp}} \tag{7.6i}$$

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM[}n\mathtt{]}\ \mathsf{num}} \tag{7.6j}$$

$$\frac{s\ \mathsf{str}}{\mathtt{LIT[}s\mathtt{]}\ \mathsf{lit}} \tag{7.6k}$$

$$\frac{s\ \mathsf{str}}{\mathtt{ID[}s\mathtt{]}\ \mathsf{id}} \tag{7.6l}$$

To emphasize the role of string concatentation, we may rewrite Rule (7.6e), for example, as follows:

$$\frac{s = s_1\,\mathtt{MUL}\,s_2\ \mathsf{str} \qquad s_1\ \mathsf{exp} \qquad s_2\ \mathsf{exp}}{s\ \mathsf{exp}}\ . \tag{7.7}$$

That is, s exp is derivable if s is the concatentation of s_1, the multiplication sign, and s_2, where s_1 exp and s_2 exp.

7.4 Ambiguity

Apart from subjective matters of readability, a principal goal of concrete syntax design is to eliminate ambiguity. The grammar of arithmetic expressions given above is *ambiguous* in the sense that some token strings may be thought of as arising in several different ways. More precisely, there are token strings s for which there is more than one derivation ending with s exp according to Rules (7.6).

For example, consider the character string 1+2*3, which, after lexical analysis, is translated to the token string

$$\mathtt{NUM[1]}\ \mathtt{ADD}\ \mathtt{NUM[2]}\ \mathtt{MUL}\ \mathtt{NUM[3]}.$$

Since string concatenation is associative, this token string can be thought of as arising in several ways, including

$$\mathtt{NUM[1]}\ \mathtt{ADD}\,{}_{\wedge}\mathtt{NUM[2]}\ \mathtt{MUL}\ \mathtt{NUM[3]}$$

and

$$\mathtt{NUM[1]}\ \mathtt{ADD}\ \mathtt{NUM[2]}\,{}_{\wedge}\,\mathtt{MUL}\ \mathtt{NUM[3]},$$

where the caret indicates the concatenation point.

One consequence of this observation is that the same token string may be seen to be grammatical according to the rules given in Section 7.3 on page 57 in two different ways. According to the first reading, the expression is principally an addition, with the first argument being a number, and the second being a multiplication of two numbers. According to the second reading, the expression is principally a multiplication, with the first argument being the addition of two numbers, and the second being a number.

Ambiguity is a *purely syntactic* property of grammars; it has nothing to do with the "meaning" of a string. For example, the token string

$$\mathtt{NUM[1]}\ \mathtt{ADD}\ \mathtt{NUM[2]}\ \mathtt{ADD}\ \mathtt{NUM[3]},$$

also admits two readings. It is immaterial that both readings have the same meaning under the usual interpretation of arithmetic expressions. Moreover, nothing prevents us from interpreting the token ADD to mean "division," in which case the two readings would hardly coincide! Nothing in the syntax itself precludes this interpretation, so we do not regard it as relevant to whether the grammar is ambiguous.

To eliminate ambiguity the grammar of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ given in Section 7.3 on page 57 must be re-structured to ensure that every grammatical string has at most one derivation according to the rules of the grammar. The main method for achieving this is to introduce precedence and associativity conventions that ensure there is only one reading of any token string. Parenthesization may be used to override these conventions, so there is no fundamental loss of expressive power in doing so.

Precedence relationships are introduced by *layering* the grammar, which is achieved by splitting syntactic classes into several sub-classes.

Factor	fct	::=	num | lit | id | LP prg RP
Term	trm	::=	fct | fct MUL trm | VB fct VB
Expression	exp	::=	trm | trm ADD exp | trm CAT exp
Program	prg	::=	exp | LET id BE exp IN prg

The effect of this grammar is to ensure that `let` has the lowest precedence, addition and concatenation intermediate precedence, and multiplication and length the highest precedence. Moreover, all forms are right-associative. Other choices of rules are possible, according to taste; this grammar illustrates one way to resolve the ambiguities of the original expression grammar.

7.5 Exercises

Chapter 8

Abstract Syntax

The concrete syntax of a language is concerned with the linear representation of the phrases of a language as strings of symbols—the form in which we write them on paper, type them into a computer, and read them from a page. The main goal of concrete syntax design is to enhance the readability and writability of the language, based on subjective criteria such as similarity to other languages, ease of editing using standard tools, and so forth.

But languages are also the subjects of study, as well as the instruments of expression. As such the concrete syntax of a language is just a nuisance. When analyzing a language mathematically we are only interested in the deep structure of its phrases, not their surface representation. The *abstract syntax* of a language exposes the hierarchical and binding structure of the language, and suppresses the linear notation used to write it on the page.

Parsing is the process of translation from concrete to abstract syntax. It consists of analyzing the linear representation of a phrase in terms of the grammar of the language and transforming it into an abstract syntax tree or an abstract binding tree that reveals the deep structure of the phrase.

8.1 Abstract Syntax Trees

The abstract syntax tree representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is specified by the following signature:

$$\begin{aligned}
\mathsf{ar}(\texttt{num}[n]) &= 0 \quad (n\ \mathsf{nat})\\
\mathsf{ar}(\texttt{str}[s]) &= 0 \quad (s\ \mathsf{str})\\
\mathsf{ar}(\texttt{id}[s]) &= 0 \quad (s\ \mathsf{str})\\
\mathsf{ar}(\texttt{plus}) &= 2\\
\mathsf{ar}(\texttt{times}) &= 2\\
\mathsf{ar}(\texttt{cat}) &= 2\\
\mathsf{ar}(\texttt{len}) &= 1\\
\mathsf{ar}(\texttt{let}[s]) &= 2
\end{aligned}$$

Observe that each identifier is regarded as operators of arity 0, and that the `let` construct is regarded as a family of operators of arity two, indexed by the identifier that it binds.

Specializing the rules for abstract syntax trees to this signature, we obtain the following inductive definition of the abstract syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$:

$$\frac{n\ \mathsf{nat}}{\texttt{num}[n]\ \mathsf{ast}} \tag{8.1a}$$

$$\frac{s\ \mathsf{str}}{\texttt{str}[s]\ \mathsf{ast}} \tag{8.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{id}[s]\ \mathsf{ast}} \tag{8.1c}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{plus}(a_1;a_2)\ \mathsf{ast}} \tag{8.1d}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{times}(a_1;a_2)\ \mathsf{ast}} \tag{8.1e}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{cat}(a_1;a_2)\ \mathsf{ast}} \tag{8.1f}$$

$$\frac{a\ \mathsf{ast}}{\texttt{len}(a)\ \mathsf{ast}} \tag{8.1g}$$

$$\frac{s\ \mathsf{id} \quad a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{let}[s](a_1;a_2)\ \mathsf{ast}} \tag{8.1h}$$

Strictly speaking, the last rule is a specialization of the rule induced by the arity assignment for `let` in which we demand that the first argument be an identifier.

8.2 Parsing Into Abstract Syntax Trees

The process of translation from concrete to abstract syntax is called *parsing*. We will define parsing as a judgement between the concrete and abstract syntax of a language. This judgement will have the mode $(\forall, \exists^{\leq 1})$ over strings and ast's, which states that the parser is a partial function of its input, being undefined for ungrammatical token strings, but otherwise uniquely determining the abstract syntax tree representation of each well-formed input.

The parsing judgements for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ follow the unambiguous grammar given in Chapter 7:

$s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{ast}$	Parse as a program
$s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}$	Parse as an expression
$s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}$	Parse as a term
$s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}$	Parse as a factor
$s\ \mathsf{num} \longleftrightarrow a\ \mathsf{ast}$	Parse as a number
$s\ \mathsf{lit} \longleftrightarrow a\ \mathsf{ast}$	Parse as a literal
$s\ \mathsf{id} \longleftrightarrow a\ \mathsf{ast}$	Parse as an identifier

These judgements are inductively defined simultaneously by the following rules:

$$\frac{n\ \mathsf{nat}}{\texttt{NUM}[n]\ \mathsf{num} \longleftrightarrow \texttt{num}[n]\ \mathsf{ast}} \tag{8.2a}$$

$$\frac{s\ \mathsf{str}}{\texttt{LIT}[s]\ \mathsf{lit} \longleftrightarrow \texttt{str}[s]\ \mathsf{ast}} \tag{8.2b}$$

$$\frac{s\ \mathsf{str}}{\texttt{ID}[s]\ \mathsf{id} \longleftrightarrow \texttt{id}[s]\ \mathsf{ast}} \tag{8.2c}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2d}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2e}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2f}$$

$$\frac{s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{ast}}{\texttt{LP}\,s\,\texttt{RP}\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2g}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2h}$$

$$\frac{s_1\ \mathsf{fct} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{trm} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\,\texttt{MUL}\,s_2\ \mathsf{trm} \longleftrightarrow \texttt{times}(a_1;a_2)\ \mathsf{ast}} \tag{8.2i}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}}{\mathtt{VB}\, s\, \mathtt{VB}\ \mathsf{trm} \longleftrightarrow \mathtt{len}(a)\ \mathsf{ast}} \tag{8.2j}$$

$$\frac{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2k}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\, \mathtt{ADD}\, s_2\ \mathsf{exp} \longleftrightarrow \mathtt{plus}(a_1;a_2)\ \mathsf{ast}} \tag{8.2l}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\, \mathtt{CAT}\, s_2\ \mathsf{exp} \longleftrightarrow \mathtt{cat}(a_1;a_2)\ \mathsf{ast}} \tag{8.2m}$$

$$\frac{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2n}$$

$$\frac{s_1\ \mathsf{id} \longleftrightarrow \mathtt{id}[s]\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast} \quad s_3\ \mathsf{prg} \longleftrightarrow a_3\ \mathsf{ast}}{\mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let}[s](a_2;a_3)\ \mathsf{ast}} \tag{8.2o}$$

A successful parse implies that the token string must have been derived according to the rules of the unambiguous grammar and that the result is a well-formed abstract syntax tree.

Theorem 8.1. *If s $\mathsf{prg} \longleftrightarrow a$ ast, then s prg and a ast, and similarly for the other parsing judgements.*

Proof. By rule induction on Rules (8.2). □

Moreover, if a string is generated according to the rules of the grammar, then it has a parse as an ast.

Theorem 8.2. *If s prg, then there is a unique a such that s $\mathsf{prg} \longleftrightarrow a$ ast, and similarly for the other parsing judgements. That is, the parsing judgements have mode $(\forall, \exists!)$ over well-formed strings and abstract syntax trees.*

Proof. By rule induction on the rules determined by reading Grammar (7.4) as an inductive definition. □

Finally, any piece of abstract syntax may be formatted as a string that parses as the given ast.

Theorem 8.3. *If a ast, then there exists a (not necessarily unique) string s such that s prg and s $\mathsf{prg} \longleftrightarrow a$ ast. That is, the parsing judgement has mode $(\exists, \forall)$.*

Proof. By rule induction on Grammar (7.4). □

The string representation of an abstract syntax tree is not unique, since we may introduce parentheses at will around any sub-expression.

8.3 Parsing Into Abstract Binding Trees

The representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ using abstract syntax trees exposes the hierarchical structure of the language, but does not manage the binding and scope of variables in a `let` expression. In this section we revise the parser given in Section 8.1 on page 62 to translate from token strings (as before) to abstract binding trees to make explicit the binding and scope of identifiers in a program.

The abstract binding tree representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is specified by the following assignment of (generalized) arities to operators:

$$\begin{aligned}
\mathsf{ar}(\texttt{num}[n]) &= ()\\
\mathsf{ar}(\texttt{str}[s]) &= ()\\
\mathsf{ar}(\texttt{plus}) &= (0,0)\\
\mathsf{ar}(\texttt{times}) &= (0,0)\\
\mathsf{ar}(\texttt{cat}) &= (0,0)\\
\mathsf{ar}(\texttt{len}) &= (0)\\
\mathsf{ar}(\texttt{let}) &= (0,1)
\end{aligned}$$

The arity of the operator `let` specifies that it takes two arguments, the second of which is an abstractor of valence 1, meaning that it binds one variable in the second argument position. Observe that identifiers are no longer declared as operators; instead, identifiers are translated by the parser into variables. Similarly, parentheses are "parsed away" on passage to abstract syntax, and thus have no representation as operators.

The revised parsing judgement, $s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt}$, between strings s and abt's a, is defined by a collection of rules similar to those given in Section 8.2 on page 63. These rules take the form of a generic inductive definition (see Chapter 2) in which the premises and conclusions of the rules involve hypothetical judgments of the form

$$\texttt{ID}[s_1]\ \mathsf{id} \longleftrightarrow x_1\ \mathsf{abt}, \ldots, \texttt{ID}[s_n]\ \mathsf{id} \longleftrightarrow x_n\ \mathsf{abt} \vdash s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt},$$

where the x_i's are pairwise distinct variable names. The hypotheses of the judgement dictate how identifiers are to be parsed as variables, for it follows from the reflexivity of the hypothetical judgement that

$$\Gamma, \texttt{ID}[s]\ \mathsf{id} \longleftrightarrow x\ \mathsf{abt} \vdash \texttt{ID}[s]\ \mathsf{id} \longleftrightarrow x\ \mathsf{abt}.$$

To maintain the association between identifiers and variables when parsing a `let` expression, we update the hypotheses to record the association between the bound identifier and a corresponding variable:

$$\frac{\begin{array}{c}\Gamma \vdash s_1\ \mathsf{id} \longleftrightarrow x\ \mathsf{abt} \qquad \Gamma \vdash s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{abt} \\ \Gamma, s_1\ \mathsf{id} \longleftrightarrow x\ \mathsf{abt} \vdash s_3\ \mathsf{prg} \longleftrightarrow a_3\ \mathsf{abt}\end{array}}{\Gamma \vdash \mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let}(a_2; x . a_3)\ \mathsf{abt}} \tag{8.3a}$$

Unfortunately, this approach does not quite work properly! If an inner `let` expression binds the same identifier as an outer `let` expression, there is an ambiguity in how to parse occurrences of that identifier. Parsing such nested `let`'s will introduce two hypotheses, say $\mathtt{ID}[s]\ \mathsf{id} \longleftrightarrow x_1\ \mathsf{abt}$ and $\mathtt{ID}[s]\ \mathsf{id} \longleftrightarrow x_2\ \mathsf{abt}$, for the same identifier $\mathtt{ID}[s]$. By the structural property of exchange, we may choose arbitrarily which to apply to any particular occurrence of $\mathtt{ID}[s]$, and hence we may parse different occurrences differently.

To rectify this we must resort to less elegant methods. Rather than use hypotheses, we instead maintain an explicit *symbol table* to record the association between identifiers and variables. We must define explicitly the procedures for creating and extending symbol tables, and for looking up an identifier in the symbol table to determine its associated variable. This gives us the freedom to implement a *shadowing* policy for re-used identifiers, according to which the most recent binding of an identifier determines the corresponding variable.

The main change to the parsing judgement is that the hypothetical judgement

$$\Gamma \vdash s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt}$$

is reduced to the categorical judgement

$$s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt}\ [\sigma],$$

where σ is a symbol table. (Analogous changes must be made to the other parsing judgements.) The symbol table is now an argument to the judgement form, rather than an implicit mechanism for performing inference under hypotheses.

The rule for parsing `let` expressions is then formulated as follows:

$$\frac{\begin{array}{cc}s_1\ \mathsf{id} \longleftrightarrow x\ [\sigma] & s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{abt}\ [\sigma] \\ \sigma' = \sigma[s_1 \mapsto x] & s_3\ \mathsf{prg} \longleftrightarrow a_3\ \mathsf{abt}\ [\sigma']\end{array}}{\mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let}(a_2; x . a_3)\ \mathsf{abt}\ [\sigma]} \tag{8.4}$$

This rule is quite similar to the hypothetical form, the difference being that we must manage the symbol table explicitly. In particular, we must include a rule for parsing identifiers, rather than relying on the reflexivity of the hypothetical judgement to do it for us.

$$\frac{\sigma(\texttt{ID[}s\texttt{]}) = x}{\texttt{ID[}s\texttt{]}\ \mathsf{id} \longleftrightarrow x\ [\sigma]} \tag{8.5}$$

The premise of this rule states that σ maps the identifier $\texttt{ID[}s\texttt{]}$ to the variable x.

Symbol tables may be defined to be finite sequences of ordered pairs of the form $(\texttt{ID[}s\texttt{]},x)$, where $\texttt{ID[}s\texttt{]}$ is an identifier and x is a variable name. Using this representation it is straightforward to define the following judgement forms:

σ symtab	well-formed symbol table
$\sigma' = \sigma[\texttt{ID[}s\texttt{]} \mapsto x]$	add new association
$\sigma(\texttt{ID[}s\texttt{]}) = x$	lookup identifier

We leave the precise definitions of these judgements as an exercise for the reader.

8.4 Syntactic Conventions

To specify a language we shall use a concise tabular notation for simultaneously specifying both its abstract and concrete syntax. Officially, the language is always a collection of abt's, but when writing examples we shall often use the concrete notation for the sake of concision and clarity. Our method of specifying the concrete syntax is sufficient for our purposes, but leaves out niggling details such as precedences of operators or the use of bracketing to disambiguate.

The method is best illustrated by example. Here is a specification of the syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ presented in the tabular style that we shall use throughout the book:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	`num`	`num`
		$\mid$	`str`	`str`
Expr	e	$::=$	x	x
		$\mid$	$\texttt{num[}n\texttt{]}$	n
		$\mid$	$\texttt{str[}s\texttt{]}$	$\texttt{"}s\texttt{"}$
		$\mid$	$\texttt{plus(}e_1\texttt{;}e_2\texttt{)}$	$e_1 + e_2$
		$\mid$	$\texttt{times(}e_1\texttt{;}e_2\texttt{)}$	$e_1 * e_2$
		$\mid$	$\texttt{cat(}e_1\texttt{;}e_2\texttt{)}$	$e_1 \text{ \^{} } e_2$
		$\mid$	$\texttt{len(}e\texttt{)}$	$\vert e \vert$
		$\mid$	$\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)}$	$\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$

This specification is to be understood as defining two judgments, τ type and τ exp, which specify two syntactic categories, one for types, the other for expressions. The abstract syntax column uses patterns ranging over abt's to determine the arities of the operators for that syntactic category. The concrete syntax column specifies the typical notational conventions used in examples. In this manner Table (8.4) defines two signatures, Ω_{type} and Ω_{expr}, that specify the operators for types and expressions, respectively. The signature for types specifies that `num` and `str` are two operators of arity $()$. The signature for expressions specifies two families of operators, $\texttt{num[}n\texttt{]}$ and $\texttt{str[}s\texttt{]}$, of arity $()$, three operators of arity $(0, 0)$ corresponding to addition, multiplication, and concatenation, one operator of arity (0) for length, and one operator of arity $(0, 1)$ for let-binding expressions to identifiers.

8.5 Exercises