Practical Foundations for Programming Languages

Second Edition

Robert Harper
Carnegie Mellon University

CAMBRIDGE
UNIVERSITY PRESS

Contents

Part VIII Partiality and Recursive Types

Part IX Dynamic Types

1 Abstract Syntax

Programming languages express computations in a form comprehensible to both people and machines. The syntax of a language specifies how various sorts of phrases (expressions, commands, declarations, and so forth) may be combined to form programs. But what are these phrases? What is a program made of?

The informal concept of syntax involves several distinct concepts. The *surface*, or *concrete*, *syntax* is concerned with how phrases are entered and displayed on a computer. The surface syntax is usually thought of as given by strings of characters from some alphabet (say, ASCII or Unicode). The *structural*, or *abstract*, *syntax* is concerned with the structure of phrases, specifically how they are composed from other phrases. At this level, a phrase is a tree, called an *abstract syntax tree*, whose nodes are operators that combine several phrases to form another phrase. The *binding* structure of syntax is concerned with the introduction and use of identifiers: how they are declared, and how declared identifiers can be used. At this level, phrases are *abstract binding trees*, which enrich abstract syntax trees with the concepts of binding and scope.

We will not concern ourselves in this book with concrete syntax but will instead consider pieces of syntax to be finite trees augmented with a means of expressing the binding and scope of identifiers within a syntax tree. To prepare the ground for the rest of the book, we define in this chapter what is a "piece of syntax" in two stages. First, we define abstract syntax trees, or ast's, which capture the hierarchical structure of a piece of syntax, while avoiding commitment to their concrete representation as a string. Second, we augment abstract syntax trees with the means of specifying the binding (declaration) and scope (range of significance) of an identifier. Such enriched forms of abstract syntax are called abstract binding trees, or abt's for short.

Several functions and relations on abt's are defined that give precise meaning to the informal ideas of binding and scope of identifiers. The concepts are infamously difficult to define properly and are the mother lode of bugs for language implementors. Consequently, precise definitions are essential, but they are also fairly technical and take some getting used to. It is probably best to skim this chapter on first reading to get the main ideas, and return to it for clarification as necessary.

1.1 Abstract Syntax Trees

An *abstract syntax tree*, or *ast* for short, is an ordered tree whose leaves are *variables*, and whose interior nodes are *operators* whose *arguments* are its children. Ast's are classified

into a variety of *sorts* corresponding to different forms of syntax. A *variable* stands for an unspecified, or generic, piece of syntax of a specified sort. Ast's can be combined by an *operator*, which has an *arity* specifying the sort of the operator and the number and sorts of its arguments. An operator of sort s and arity $s_1, \ldots, s_n$ combines $n \geq 0$ ast's of sort $s_1, \ldots, s_n$, respectively, into a compound ast of sort s.

The concept of a variable is central and therefore deserves special emphasis. A variable is an *unknown* object drawn from some domain. The unknown can become known by *substitution* of a particular object for all occurrences of a variable in a formula, thereby specializing a general formula to a particular instance. For example, in school algebra variables range over real numbers, and we may form polynomials, such as $x^2 + 2\,x + 1$, that can be specialized by substitution of, say, 7 for x to obtain $7^2 + (2 \times 7) + 1$, which can be simplified according to the laws of arithmetic to obtain 64, which is $(7 + 1)^2$.

Abstract syntax trees are classified by *sorts* that divide ast's into syntactic categories. For example, familiar programming languages often have a syntactic distinction between expressions and commands; these are two sorts of abstract syntax trees. Variables in abstract syntax trees range over sorts in the sense that only ast's of the specified sort of the variable can be plugged in for that variable. Thus, it would make no sense to replace an expression variable by a command, nor a command variable by an expression, the two being different sorts of things. But the core idea carries over from school mathematics, namely that *a variable is an unknown, or a place-holder, whose meaning is given by substitution*.

As an example, consider a language of arithmetic expressions built from numbers, addition, and multiplication. The abstract syntax of such a language consists of a single sort Exp generated by these operators:

1. An operator num[n] of sort Exp for each $n \in \mathbb{N}$.
2. Two operators, plus and times, of sort Exp, each with two arguments of sort Exp.

The expression $2 + (3 \times x)$, which involves a variable, x, would be represented by the ast

$$\texttt{plus}(\texttt{num}[2]; \texttt{times}(\texttt{num}[3]; x))$$

of sort Exp, under the assumption that x is also of this sort. Because, say, num[4], is an ast of sort Exp, we may plug it in for x in the above ast to obtain the ast

$$\texttt{plus}(\texttt{num}[2]; \texttt{times}(\texttt{num}[3]; \texttt{num}[4])),$$

which is written informally as $2 + (3 \times 4)$. We may, of course, plug in more complex ast's of sort Exp for x to obtain other ast's as result.

The tree structure of ast's provides a very useful principle of reasoning, called *structural induction*. Suppose that we wish to prove that some property $\mathcal{P}(a)$ holds for all ast's a of a given sort. To show this, it is enough to consider all the ways in which a can be generated and show that the property holds in each case under the assumption that it holds for its constituent ast's (if any). So, in the case of the sort Exp just described, we must show

1. The property holds for any variable x of sort Exp: prove that $\mathcal{P}(x)$.
2. The property holds for any number, num[n]: for every $n \in \mathbb{N}$, prove that $\mathcal{P}(\texttt{num}[n])$.

3. Assuming that the property holds for a_1 and a_2, prove that it holds for $\texttt{plus}(a_1; a_2)$ and $\texttt{times}(a_1; a_2)$: if $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$, then $\mathcal{P}(\texttt{plus}(a_1; a_2))$ and $\mathcal{P}(\texttt{times}(a_1; a_2))$.

Because these cases exhaust all possibilities for the formation of a, we are assured that $\mathcal{P}(a)$ holds for any ast a of sort Exp.

It is common to apply the principle of structural induction in a form that takes account of the interpretation of variables as place-holders for ast's of the appropriate sort. Informally, it is often useful to prove a property of an ast involving variables in a form that is conditional on the property holding for the variables. Doing so anticipates that the variables will be replaced with ast's that ought to have the property assumed for them, so that the result of the replacement will have the property as well. This amounts to applying the principle of structural induction to properties $\mathcal{P}(a)$ of the form "if a involves variables $x_1, \ldots, x_k$, and $\mathcal{Q}$ holds of each x_i, then $\mathcal{Q}$ holds of a," so that a proof of $\mathcal{P}(a)$ for all ast's a by structural induction is just a proof that $\mathcal{Q}(a)$ holds for all ast's a under the assumption that $\mathcal{Q}$ holds for its variables. When there are no variables, there are no assumptions, and the proof of $\mathcal{P}$ is a proof that $\mathcal{Q}$ holds for all *closed* ast's. On the other hand, if x is a variable in a, and we replace it by an ast b for which $\mathcal{Q}$ holds, then $\mathcal{Q}$ will hold for the result of replacing x by b in a.

For the sake of precision, we now give precise definitions of these concepts. Let $\mathcal{S}$ be a finite set of sorts. For a given set $\mathcal{S}$ of sorts, an *arity* has the form $(s_1, \ldots, s_n)s$, which specifies the sort $s \in \mathcal{S}$ of an operator taking $n \geq 0$ arguments, each of sort $s_i \in \mathcal{S}$. Let $\mathcal{O} = \{\mathcal{O}_\alpha\}$ be an arity-indexed family of disjoint sets of *operators* $\mathcal{O}_\alpha$ of arity α. If o is an operator of arity $(s_1, \ldots, s_n)s$, we say that o has sort s and has n arguments of sorts $s_1, \ldots, s_n$.

Fix a set $\mathcal{S}$ of sorts and an arity-indexed family $\mathcal{O}$ of sets of operators of each arity. Let $\mathcal{X} = \{\mathcal{X}_s\}_{s \in \mathcal{S}}$ be a sort-indexed family of disjoint finite sets $\mathcal{X}_s$ of *variables* x of sort s. When $\mathcal{X}$ is clear from context, we say that a variable x is of sort s if $x \in \mathcal{X}_s$, and we say that x is *fresh for* $\mathcal{X}$, or just *fresh* when $\mathcal{X}$ is understood, if $x \notin \mathcal{X}_s$ for any sort s. If x is fresh for $\mathcal{X}$ and s is a sort, then $\mathcal{X}, x$ is the family of sets of variables obtained by adding x to $\mathcal{X}_s$. The notation is ambiguous in that the sort s is not explicitly stated but determined from context.

The family $\mathcal{A}[\mathcal{X}] = \{\mathcal{A}[\mathcal{X}]_s\}_{s \in \mathcal{S}}$ of *abstract syntax trees*, or *ast's*, of sort s is the smallest family satisfying the following conditions:

1. A variable of sort s is an ast of sort s: if $x \in \mathcal{X}_s$, then $x \in \mathcal{A}[\mathcal{X}]_s$.
2. Operators combine ast's: if o is an operator of arity $(s_1, \ldots, s_n)s$, and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}$, $\ldots$, $a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]_s$.

It follows from this definition that the principle of *structural induction* can be used to prove that some property $\mathcal{P}$ holds of every ast. To show $\mathcal{P}(a)$ holds for every $a \in \mathcal{A}[\mathcal{X}]$, it is enough to show:

1. If $x \in \mathcal{X}_s$, then $\mathcal{P}_s(x)$.
2. If o has arity $(s_1, \ldots, s_n)s$ and $\mathcal{P}_{s_1}(a_1)$ and $\ldots$ and $\mathcal{P}_{s_n}(a_n)$, then $\mathcal{P}_s(o(a_1; \ldots; a_n))$.

For example, it is easy to prove by structural induction that $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$ whenever $\mathcal{X} \subseteq \mathcal{Y}$.

Variables are given meaning by *substitution*. If $a \in \mathcal{A}[\mathcal{X}, x]_{s'}$, and $b \in \mathcal{A}[\mathcal{X}]_s$, then $[b/x]a \in \mathcal{A}[\mathcal{X}]_{s'}$ is the result of substituting b for every occurrence of x in a. The ast a is called the *target*, and x is called the *subject*, of the substitution. Substitution is defined by the following equations:

1. $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(a_1; \ldots; a_n) = o([b/x]a_1; \ldots; [b/x]a_n)$.

For example, we may check that

$$[\texttt{num}[2]/x]\texttt{plus}(x; \texttt{num}[3]) = \texttt{plus}(\texttt{num}[2]; \texttt{num}[3]).$$

We may prove by structural induction that substitution on ast's is well-defined.

Theorem 1.1. *If $a \in \mathcal{A}[\mathcal{X}, x]$, then for every $b \in \mathcal{A}[\mathcal{X}]$ there exists a unique $c \in \mathcal{A}[\mathcal{X}]$ such that $[b/x]a = c$*

Proof By structural induction on a. If $a = x$, then $c = b$ by definition; otherwise, if $a = y \neq x$, then $c = y$, also by definition. Otherwise, $a = o(a_1, \ldots, a_n)$, and we have by induction unique $c_1, \ldots, c_n$ such that $[b/x]a_1 = c_1$ and $\ldots$ $[b/x]a_n = c_n$, and so c is $c = o(c_1; \ldots; c_n)$, by definition of substitution. □

1.2 Abstract Binding Trees

Abstract binding trees, or *abt's*, enrich ast's with the means to introduce new variables and symbols, called a *binding*, with a specified range of significance, called its *scope*. The scope of a binding is an abt within which the bound identifier can be used, either as a place-holder (in the case of a variable declaration) or as the index of some operator (in the case of a symbol declaration). Thus, the set of active identifiers can be larger within a subtree of an abt than it is within the surrounding tree. Moreover, different subtrees may introduce identifiers with disjoint scopes. The crucial principle is that any use of an identifier should be understood as a reference, or abstract pointer, to its binding. One consequence is that the choice of identifiers is immaterial, so long as we can always associate a unique binding with each use of an identifier.

As a motivating example, consider the expression $\texttt{let}\, x\, \texttt{be}\, a_1\, \texttt{in}\, a_2$, which introduces a variable x for use within the expression a_2 to stand for the expression a_1. The variable x is bound by the `let` expression for use within a_2; any use of x within a_1 refers to a different variable that happens to have the same name. For example, in the expression $\texttt{let}\, x\, \texttt{be}\, 7\, \texttt{in}\, x + x$ occurrences of x in the addition refer to the variable introduced by the `let`. On the other hand, in the expression $\texttt{let}\, x\, \texttt{be}\, x * x\, \texttt{in}\, x + x$, occurrences of x within the multiplication refer to a different variable than those occurring within the addition. The

latter occurrences refer to the binding introduced by the `let`, whereas the former refer to some outer binding not displayed here.

The names of bound variables are immaterial insofar as they determine the same binding. So, for example, $\texttt{let}\, x\, \texttt{be}\, x * x\, \texttt{in}\, x + x$ could just as well have been written $\texttt{let}\, y\, \texttt{be}\, x * x\, \texttt{in}\, y + y$, without changing its meaning. In the former case, the variable x is bound within the addition, and in the latter, it is the variable y, but the "pointer structure" remains the same. On the other hand, the expression $\texttt{let}\, x\, \texttt{be}\, y * y\, \texttt{in}\, x + x$ has a different meaning to these two expressions, because now the variable y within the multiplication refers to a different surrounding variable. Renaming of bound variables is constrained to the extent that it must not alter the reference structure of the expression. For example, the expression

$$\texttt{let}\, x\, \texttt{be}\, 2\, \texttt{in}\, \texttt{let}\, y\, \texttt{be}\, 3\, \texttt{in}\, x + x$$

has a different meaning than the expression

$$\texttt{let}\, y\, \texttt{be}\, 2\, \texttt{in}\, \texttt{let}\, y\, \texttt{be}\, 3\, \texttt{in}\, y + y,$$

because the y in the expression $y + y$ in the second case refers to the inner declaration, not the outer one as before.

The concept of an ast can be enriched to account for binding and scope of a variable. These enriched ast's are called *abstract binding trees*, or *abt's* for short. Abt's generalize ast's by allowing an operator to bind any finite number (possibly zero) of variables in each argument. An argument to an operator is called an *abstractor* and has the form $x_1, \ldots, x_k.a$. The sequence of variables $x_1, \ldots, x_k$ are bound within the abt a. (When k is zero, we elide the distinction between $.a$ and a itself.) Written in the form of an abt, the expression $\texttt{let}\, x\, \texttt{be}\, a_1\, \texttt{in}\, a_2$ has the form $\texttt{let}(a_1; x.a_2)$, which more clearly specifies that the variable x is bound within a_2, and not within a_1. We often write $\vec{x}$ to stand for a finite sequence $x_1, \ldots, x_n$ of distinct variables and write $\vec{x}.a$ to mean $x_1, \ldots, x_n.a$.

To account for binding, operators are assigned *generalized arities* of the form $(\upsilon_1, \ldots, \upsilon_n)s$, which specifies operators of sort s with n arguments of *valence* $\upsilon_1, \ldots, \upsilon_n$. In general a valence υ has the form $s_1, \ldots, s_k.s$, which specifies the sort of an argument as well as the number and sorts of the variables bound within it. We say that a sequence $\vec{x}$ of variables is of sort $\vec{s}$ to mean that the two sequences have the same length k and that the variable x_i is of sort s_i for each $1 \leq i \leq k$.

Thus, to specify that the operator `let` has arity (Exp, Exp.Exp)Exp indicates that it is of sort Exp whose first argument is of sort Exp and binds no variables and whose second argument is also of sort Exp and within which is bound one variable of sort Exp. The informal expression $\texttt{let}\, x\, \texttt{be}\, 2 + 2\, \texttt{in}\, x \times x$ may then be written as the abt

$$\texttt{let}(\texttt{plus}(\texttt{num}[2]; \texttt{num}[2]); x.\texttt{times}(x; x))$$

in which the operator `let` has two arguments, the first of which is an expression, and the second of which is an abstractor that binds one expression variable.

Fix a set $\mathcal{S}$ of sorts and a family $\mathcal{O}$ of disjoint sets of operators indexed by their generalized arities. For a given family of disjoint sets of variables $\mathcal{X}$, the family of *abstract binding*

trees, or *abt's* $\mathcal{B}[\mathcal{X}]$, is defined similarly to $\mathcal{A}[\mathcal{X}]$, except that $\mathcal{X}$ is not fixed throughout the definition but rather changes as we enter the scopes of abstractors.

This simple idea is surprisingly hard to make precise. A first attempt at the definition is as the least family of sets closed under the following conditions:

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.
2. For each operator o of arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)s$, if $a_1 \in \mathcal{B}[\mathcal{X}, \vec{x}_1]_{s_1}$, $\ldots$, and $a_n \in \mathcal{B}[\mathcal{X}, \vec{x}_n]_{s_n}$, then $o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The bound variables are adjoined to the set of active variables within each argument, with the sort of each variable determined by the valence of the operator.

This definition is *almost* correct but fails to properly account for renaming of bound variables. An abt of the form $\mathtt{let}(a_1; x.\mathtt{let}(a_2; x.a_3))$ is ill-formed according to this definition, because the first binding adds x to $\mathcal{X}$, which implies that the second cannot also add x to $\mathcal{X}, x$, because it is not fresh for $\mathcal{X}, x$. The solution is to ensure that each of the arguments is well-formed regardless of the choice of bound variable names, which is achieved using *fresh renamings*, which are bijections between sequences of variables. Specifically, a fresh renaming (relative to $\mathcal{X}$) of a finite sequence of variables $\vec{x}$ is a bijection $\rho : \vec{x} \leftrightarrow \vec{x}'$ between $\vec{x}$ and $\vec{x}'$, where $\vec{x}'$ is fresh for $\mathcal{X}$. We write $\widehat{\rho}(a)$ for the result of replacing each occurrence of x_i in a by $\rho(x_i)$, its fresh counterpart.

This is achieved by altering the second clause of the definition of abt's using fresh renamings as follows:

> For each operator o of arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)s$, if for each $1 \leq i \leq n$ and each fresh renaming $\rho_i : \vec{x}_i \leftrightarrow \vec{x}_i'$, we have $\widehat{\rho_i}(a_i) \in \mathcal{B}[\mathcal{X}, \vec{x}_i']$, then $o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The renaming $\widehat{\rho_i}(a_i)$ of each a_i ensures that collisions cannot occur and that the abt is valid for almost all renamings of any bound variables that occur within it.

The principle of structural induction extends to abt's and is called *structural induction modulo fresh renaming*. It states that to show that $\mathcal{P}[\mathcal{X}](a)$ holds for every $a \in \mathcal{B}[\mathcal{X}]$, it is enough to show the following:

1. if $x \in \mathcal{X}_s$, then $\mathcal{P}[\mathcal{X}]_s(x)$.
2. For every o of arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)s$, if for each $1 \leq i \leq n$, $\mathcal{P}[\mathcal{X}, \vec{x}_i']_{s_i}(\widehat{\rho_i}(a_i))$ holds for every $\rho_i : \vec{x}_i \leftrightarrow \vec{x}_i'$ with $\vec{x}_i' \notin \mathcal{X}$, then $\mathcal{P}[\mathcal{X}]_s(o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n))$.

The second condition ensures that the inductive hypothesis holds for *all* fresh choices of bound variable names, and not just the ones actually given in the abt.

As an example let us define the judgment $x \in a$, where $a \in \mathcal{B}[\mathcal{X}, x]$, to mean that x *occurs free* in a. Informally, this means that x is bound somewhere outside of a, rather than within a itself. If x is bound within a, then those occurrences of x are different from those occurring outside the binding. The following definition ensures that this is the case:

1. $x \in x$.
2. $x \in o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n)$ if there exists $1 \leq i \leq n$ such that for every fresh renaming $\rho : \vec{x}_i \leftrightarrow \vec{z}_i$ we have $x \in \widehat{\rho}(a_i)$.

The first condition states that x is free in x but not free in y for any variable y other than x. The second condition states that if x is free in some argument, independently of the choice of bound variable names in that argument, then it is free in the overall abt.

The relation $a =_\alpha b$ of *α-equivalence* (so-called for historical reasons) means that a and b are identical up to the choice of bound variable names. The α-equivalence relation is the strongest congruence containing the following two conditions:

1. $x =_\alpha x$.
2. $o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n) =_\alpha o(\vec{x}'_1.a'_1; \ldots; \vec{x}'_n.a'_n)$ if for every $1 \leq i \leq n$, $\widehat{\rho_i}(a_i) =_\alpha \widehat{\rho'_i}(a'_i)$ for all fresh renamings $\rho_i : \vec{x}_i \leftrightarrow \vec{z}_i$ and $\rho'_i : \vec{x}'_i \leftrightarrow \vec{z}_i$.

The idea is that we rename $\vec{x}_i$ and $\vec{x}'_i$ consistently, avoiding confusion, and check that a_i and a'_i are α-equivalent. If $a =_\alpha b$, then a and b are *α-variants* of each other.

Some care is required in the definition of *substitution* of an abt b of sort s for free occurrences of a variable x of sort s in some abt a of some sort, written $[b/x]a$. Substitution is partially defined by the following conditions:

1. $[b/x]x = b$, and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n) = o(\vec{x}_1.a'_1; \ldots; \vec{x}_n.a'_n)$, where, for each $1 \leq i \leq n$, we require that $\vec{x}_i \notin b$, and we set $a'_i = [b/x]a_i$ if $x \notin \vec{x}_i$, and $a'_i = a_i$ otherwise.

The definition of $[b/x]a$ is quite delicate and merits careful consideration.

One trouble spot for substitution is to notice that if x is bound by an abstractor within a, then x does not occur free within the abstractor and hence is unchanged by substitution. For example, $[b/x]\texttt{let}(a_1; x.a_2) = \texttt{let}([b/x]a_1; x.a_2)$, there being no free occurrences of x in $x.a_2$. Another trouble spot is the *capture* of a free variable of b during substitution. For example, if $y \in b$ and $x \neq y$, then $[b/x]\texttt{let}(a_1; y.a_2)$ is undefined, rather than being $\texttt{let}([b/x]a_1; y.[b/x]a_2)$, as one might at first suspect. For example, provided that $x \neq y$, $[y/x]\texttt{let}(\texttt{num}[0]; y.\texttt{plus}(x; y))$ is undefined, not $\texttt{let}(\texttt{num}[0]; y.\texttt{plus}(y; y))$, which confuses two different variables named y.

Although capture avoidance is an essential characteristic of substitution, it is, in a sense, merely a technical nuisance. If the names of bound variables have no significance, then capture can always be avoided by first renaming the bound variables in a to avoid any free variables in b. In the foregoing example, if we rename the bound variable y to y' to obtain $a' \triangleq \texttt{let}(\texttt{num}[0]; y'.\texttt{plus}(x; y'))$, then $[b/x]a'$ *is* defined and is equal to $\texttt{let}(\texttt{num}[0]; y'.\texttt{plus}(b; y'))$. The price for avoiding capture in this way is that substitution is only determined up to α-equivalence, and so we may no longer think of substitution as a function but only as a proper relation.

To restore the functional character of substitution, it is sufficient to adopt the *identification convention*, which is stated as follows:

Abstract binding trees are always identified up to α-equivalence.

That is, α-equivalent abt's are regarded as identical. Substitution can be extended to α-equivalence classes of abt's to avoid capture by choosing representatives of the equivalence classes of b and a in such a way that substitution is defined, then forming the equivalence class of the result. Any two choices of representatives for which substitution is defined gives α-equivalent results, so that substitution becomes a well-defined total function. *We will adopt the identification convention for abt's throughout this book.*

It will often be necessary to consider languages whose abstract syntax cannot be specified by a fixed set of operators but rather requires that the available operators be sensitive to the context in which they occur. For our purposes, it will suffice to consider a set of *symbolic parameters*, or *symbols*, that index families of operators so that as the set of symbols varies, so does the set of operators. An *indexed operator* o is a family of operators indexed by symbols u, so that $o[u]$ is an operator when u is an available symbol. If $\mathcal{U}$ is a finite set of symbols, then $\mathcal{B}[\mathcal{U};\mathcal{X}]$ is the sort-indexed family of abt's that are generated by operators and variables as before, admitting all indexed operator instances by symbols $u \in \mathcal{U}$. Whereas a variable is a place-holder that stands for an unknown abt of its sort, a symbol *does not stand for anything*, and is not, itself, an abt. The only significance of symbol is whether it is the same as or differs from another symbol; the operator instances $o[u]$ and $o[u']$ are the same exactly when u is u' and are the same symbol.

The set of symbols is extended by introducing a *new*, or *fresh*, symbol within a scope using the abstractor $u.a$, which binds the symbol u within the abt a. An abstracted symbol is "new" in the same sense as for an abstracted variable: the name of the bound symbol can be varied at will provided that no conflicts arise. This renaming property ensures that an abstracted symbol is distinct from all others in scope. The only difference between symbols and variables is that the only operation on symbols is renaming; there is no notion of substitution for a symbol.

Finally, a word about notation: to help improve the readability we often "group" and "stage" the arguments to an operator, using round brackets and braces to show grouping, and generally regarding stages to progress from right to left. All arguments in a group are considered to occur at the same stage, though their order is significant, and successive groups are considered to occur in sequential stages. Staging and grouping is often a helpful mnemonic device, but has no fundamental significance. For example, the abt $o\{a_1;a_2\}(a_3;x.a_4)$ is the same as the abt $o(a_1;a_2;a_3;x.a_4)$, as would be any other order-preserving grouping or staging of its arguments.

1.3 Notes

The concept of abstract syntax has its origins in the pioneering work of Church, Turing, and Gödel, who first considered writing programs that act on representations of programs.

Originally, programs were represented by natural numbers, using encodings, now called *Gödel-numberings*, based on the prime factorization theorem. Any standard text on mathematical logic, such as Kleene (1952), has a thorough account of such representations. The Lisp language (McCarthy, 1965; Allen, 1978) introduced a much more practical and direct representation of syntax as *symbolic expressions*. These ideas were developed further in the language ML (Gordon et al., 1979), which featured a type system capable of expressing abstract syntax trees. The AUTOMATH project (Nederpelt et al., 1994) introduced the idea of using Church's λ notation (Church, 1941) to account for the binding and scope of variables. These ideas were developed further in LF (Harper et al., 1993).

The concept of abstract binding trees presented here was inspired by the system of notation developed in the NuPRL Project, which is described in Constable (1986) and from Martin-Löf's system of arities, which is described in Nordstrom et al. (1990). Their enrichment with symbol binders is influenced by Pitts and Stark (1993).

Exercises

1.1. Prove by structural induction on abstract syntax trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

1.2. Prove by structural induction modulo renaming on abstract binding trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{B}[\mathcal{X}] \subseteq \mathcal{B}[\mathcal{Y}]$.

1.3. Show that if $a =_\alpha a'$ and $b =_\alpha b'$ and both $[b/x]a$ and $[b'/x]a'$ are defined, then $[b/x]a =_\alpha [b'/x]a'$.

1.4. Bound variables can be seen as the formal analogs of pronouns in natural languages. The binding occurrence of a variable at an abstractor fixes a "fresh" pronoun for use within its body that refers unambiguously to that variable (in contrast to English, in which the referent of a pronoun can often be ambiguous). This observation suggests an alternative representation of abt's, called *abstract binding graphs*, or *abg's* for short, as directed graphs constructed as follows:

(a) Free variables are atomic nodes with no outgoing edges.
(b) Operators with n arguments are n-ary nodes, with one outgoing edge directed at each of their children.
(c) Abstractors are nodes with one edge directed to the scope of the abstracted variable.
(d) Bound variables are back edges directed at the abstractor that introduced it.

Notice that ast's, thought of as abt's with no abstractors, are *acyclic* directed graphs (more precisely, variadic trees), whereas general abt's can be *cyclic*. Draw a few examples of abg's corresponding to the example abt's given in this chapter. Give a precise definition of the sort-indexed family $\mathcal{G}[\mathcal{X}]$ of abstract binding graphs. What representation would you use for bound variables (back edges)?