Computer Science 15-312
Concrete and Abstract Syntax

Robert Harper

Spring, 2000

1 Introduction

The *concrete syntax* of a language consists of the rules for representing expressions as strings, linear sequences of characters (or symbols) 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 (grounded in the theory of formal languages) for eliminating ambiguity, readability is, of course, a matter of taste on which reasonable people may disagree.

Techniques for enhancing readability include the use of suggestive key words and phrases, and punctuation and layout conventions. Techniques for eliminating ambiguity include precedence conventions for binary operators and various forms of parentheses for grouping sub-expressions. You are no doubt familiar with all of these ideas; we will have little more to say about concrete syntax design in this course.

The *abstract syntax* of a language consists of rules defining the fundamental constructs of the language thought of as elements of an inductively-defined set of *abstract syntax trees* or *terms*. The abstract syntax of a language avoids the complications inherent in the representation of expressions as strings. The translation from concrete to abstract syntax is called *parsing*. The job of the parser is to determine the underlying syntactic structure of an expression represented as a string. There can be many different concrete syntaxes for a given abstract syntax (and hence many parsers, one for each concrete syntax), but in practice we normally fix one concrete and one abstract syntax for a language.

2 Grammars

The standard method for specifying the syntax of a language is a *(context-free) grammar*. A grammar consists of a set of *syntactic categories* whose elements are determined by a set of *productions*. A production may be thought of as a recursive specification of a set of the constructs of a syntactic category described in terms of the elements of some other, or even the very same, syntactic category.

An example will explain everything. Let us give a grammar defining the syntax of a tiny fragment of an ML-like language in which we have both expressions and declarations. Here's the grammar:

Category	*Member*		*Constructs*
Variables	x	$::=$	$v_1 \mid v_2 \mid \cdots$
Numbers	n	$::=$	$\mathtt{0} \mid \mathtt{S}(n)$
Expressions	e	$::=$	$x \mid n \mid e_1\mathtt{+}e_2 \mid e_1\mathtt{*}e_2 \mid \mathtt{let}\,d\,\mathtt{in}\,e$
Declarations	d	$::=$	$x\mathtt{=}e \mid d_1\,\mathtt{and}\,d_2$

The symbol "$::=$" may be read as "is" and the symbol "$\mid$" may be read as "or". Thus a variable x is one of an infinite enumeration v_1, v_2, $\ldots$; a number n is a base-1 numeral for a natural number, an expression; e is either a variable x, a number n, the sum or product of two expressions, or a `let` expression; and a declaration d is either a variable binding or a parallel composition of declarations. Each alternative is a production generating an element of the category on the left of the "$::=$" symbol.

The purpose of a grammar is to determine the elements of its syntactic categories. But what are those elements? We can read the grammar in two ways. Under one reading the elements of the syntactic categories are strings (sequences of characters) determined by the productions of the grammar. For example, the string

```
S(0) + S(S(0)) * S(S(S(0)))
```

is an element of the syntactic category of expressions, according to the grammar given above. In the absence of parenthesization or precedence conventions (which are not specified in the grammar) this string is ambiguous; it can be read in one of two ways, either as a sum of a number and a product, or as the product of a sum and a number. The job of the concrete syntax designer is to "beef up" the grammar with enough additional constructs to eliminate these ambiguities (for example, by introducing a precedence relation among binary operators) and to enhance the readability (for example, by admitting ordinary base-10 notation for numbers).

Another reading of the grammar is as a specification of the abstract syntax of the language. Each syntactic category consists of a set of structured *terms*, also known as *abstract syntax trees*, or *ast's*. Under this reading the keywords in the grammar stand for constructors that build ast's out of other ast's. For example, the following term is an element of the category of expressions.

```
+(S(0),*(S(S(0)),S(S(S(0))))),
```

as is the following, different, term

```
*(+(S(0),S(S(0))),S(S(S(0)))).
```

In both cases we do not think of the syntactic elements as strings of characters, but rather as trees whose nodes are labelled with constructors (such as `+`, `*`, `S`, and `0`) and whose children are written as arguments of the constructors. The advantage of this presentation is that there is no ambiguity about how to "read" an element of a category — the term structure specifies the "deep structure" of the expression without recourse to any additional conventions or markings.

It is straightforward to represent the abstract syntax of a language in ML. Each syntactic category is represented by a recursive `datatype` whose constructors are generated by the productions of the grammar. Here is the representation of the above grammar (read as an abstract syntax specification) in ML:

```
datatype
    Var = V of int
and Num = Zero | Succ of Num
and Exp = Var of Var | Num of Num | Plus of Exp * Exp
        | Times of Exp * Exp | Let of Dec * Exp
and Dec = Bind of Var * Exp | And of Dec * Dec
```

The correspondence with the grammar is vivid. We need only choose a value constructor to correspond to the right-hand side of each production, and transliterate the productions into ML notation.

3 Grammars as Inductive Definitions

The elements of the syntactic categories specified by a grammar may be defined by interpreting the grammar as an inductive definition. The overall idea is to regard each production as a rule, and to take the meaning of the grammar to be the set inductively defined by these rules. What's interesting is that we can recover both interpretations of the grammar simply by changing the meaning of the constructors used in the inductive definition.

There is one complication in rendering a grammar as an inductive definition. Whereas the grammar notation defines *several* sets (one for each category), our discussion of inductive definitions covers only the definition of a *single* set by a set of rules. We can easily generalize the theory of inductive definitions to the simultaneous inductive definition of several sets. We need only "mark" the elements with an indication of the set to which they belong so that we can sort out which rules define elements of which set.

Here is the grammar of the preceding section presented as an inductive definition:

$$\frac{}{v_0\ \mathsf{var}} \quad \frac{}{v_1\ \mathsf{var}} \quad \cdots$$

$$\frac{}{Z\ \mathsf{num}} \quad \frac{n\ \mathsf{num}}{S(n)\ \mathsf{num}}$$

$$\frac{x\ \mathsf{var}}{var(x)\ \mathsf{exp}} \qquad \frac{n\ \mathsf{num}}{num(n)\ \mathsf{exp}}$$

$$\frac{e_1\ \mathsf{exp} \qquad e_2\ \mathsf{exp}}{plus(e_1, e_2)\ \mathsf{exp}} \quad \frac{e_1\ \mathsf{exp} \qquad e_2\ \mathsf{exp}}{times(e_1, e_2)\ \mathsf{exp}}$$

$$\frac{d\ \mathsf{dec} \qquad e\ \mathsf{exp}}{let(d, e)\ \mathsf{exp}}$$

$$\frac{x\ \mathsf{var} \qquad e\ \mathsf{exp}}{bind(x, e)\ \mathsf{dec}} \quad \frac{d_1\ \mathsf{dec} \qquad d_2\ \mathsf{dec}}{and(d_1, d_2)\ \mathsf{dec}}$$

Notice that we mark each object with its syntactic category. For example, the category of expressions consists of those objects e for which $e\ \mathsf{exp}$ is an element of the set inductively defined by the foregoing rules.

To complete the interpretation we must define the objects and constructors used in the rules. If we wish to regard the elements of the categories as strings, we may define these as follows:

$$
\begin{array}{rcl}
v_0 & = & \texttt{"v0"} \\
v_1 & = & \texttt{"v1"} \\
\vdots & & \\
& & \\
Z & = & \texttt{"0"} \\
S(s) & = & \texttt{"S("}\ \hat{}\ s\ \hat{}\ \texttt{")"} \\
& & \\
\mathit{var}(s) & = & s \\
\mathit{num}(s) & = & s \\
\mathit{plus}(s_1, s_2) & = & s_1\ \hat{}\ \texttt{"+"}\ \hat{}\ s_2 \\
\mathit{times}(s_1, s_2) & = & s_1\ \hat{}\ \texttt{"*"}\ \hat{}\ s_2 \\
\mathit{let}(s_d, s_e) & = & \texttt{"let "}\ \hat{}\ s_d\ \hat{}\ \texttt{" in "}\ \hat{}\ s_e \\
& & \\
\mathit{bind}(s_x, s_e) & = & s_x\ \hat{}\ \texttt{"="}\ \hat{}\ s_e \\
\mathit{and}(s_1, s_2) & = & s_1\ \hat{}\ \texttt{" and "}\ \hat{}\ s_2
\end{array}
$$

Here we use an ML-like notation for strings.

If we wish to regard the grammar as defining abstract syntax trees, we may define these as follows:

$$
\begin{array}{rcl}
v_0 & = & \texttt{V(0)} \\
v_1 & = & \texttt{V(1)} \\
\vdots & & \\
& & \\
Z & = & \texttt{Zero} \\
S(t) & = & \texttt{Succ(}t\texttt{)} \\
& & \\
\mathit{var}(v) & = & \texttt{Var(}v\texttt{)} \\
\mathit{num}(n) & = & \texttt{Num(}n\texttt{)} \\
\mathit{plus}(t_1, t_2) & = & \texttt{Plus(}t_1\texttt{,}t_2\texttt{)} \\
\mathit{times}(t_1, t_2) & = & \texttt{Times(}t_1\texttt{,}t_2\texttt{)} \\
\mathit{let}(t_d, t_e) & = & \texttt{Let(}t_d\texttt{,}t_e\texttt{)} \\
& & \\
\mathit{bind}(t_x, t_e) & = & \texttt{Bind(}t_x\texttt{,}t_e\texttt{)} \\
\mathit{and}(t_1, t_2) & = & \texttt{And(}t_1\texttt{,}t_2\texttt{)}
\end{array}
$$

Here we use the ML `datatype` constructors given earlier to build up the abstract syntax. What is important is that the ranges of distinct constructors are disjoint and that the constructors are injective (1:1) so that a term has a unique outermost constructor from which we can recover its constituent terms.

4 Structural Induction

Rendering a grammar as an inductive definition allows us to apply the general principle of rule induction to abstract syntax definitions. This is called *proof by structural induction*. The principal of structural induction for expressions states that to prove that a property P holds of every expression e, it is enough to prove

1. P holds of every variable;
2. P holds of every number;
3. If P holds of e_1 and e_2, then P holds of e_1`+`e_2, and similarly for `Times`.
4. If P holds of e, then P holds of `let` d `in` e.

In practice it is usually necessary to extend P to the other syntactic categories, as follows. To prove that P holds of a declaration d it is enough to show that:

1. If P holds of x and e, then P holds of x`=`e.
2. If P holds of d_1 and d_2, then P holds of d_1 `and` d_2.

Similarly, P holds of every variable v iff P holds of v_i for every $i \geq 0$, and P holds of every number n iff

1. P holds of `Zero`;
2. If P holds of t, then P holds of `Succ(`t`)`.

We then change the clause for `Let` to

4' If P holds of d and e, then P holds of `let` d `in` e.

Exercise 1
Define the depth *of an expression and prove that every expression e has a unique depth by induction on the structure of e.* Hint: *First give conditions relating the depth of each form of expression to the depth of its constituent expressions. You will need to define a notion of depth for declarations.*

5 Binding and Scope

Abstract syntax allows us to separate out what is common to many languages from what is specific to each of them. While well-chosen lexical conventions are crucial for human readers, they are not important for programming or proving properties of languages. The abstract syntax of a phrase reveals its "deep structure" and sheds the niceties designed for human readers. The job of a parser is to translate from concrete to abstract syntax, resolving ambiguities and reducing an expression to its barest representation.

Abstract syntax trees are a natural way to capture the deep structure of a phrase since it delineates the hierarchical relationships among the components of that phrase. For example, the abstract syntax of an English sentence might indicate by its structure that the sentence is a conjunction of two independent clauses, each of which starts with a dependent clause, and so forth. Of course it is not a trivial matter to determine the abstract syntax of a sentence of natural language, but people are pretty good at it and computers are getting much better.

One aspect shared by many programming languages is the notion of *binding* and *scope* of *variables*. For example, in the expression `let` x=e `in` e' the variable x is *bound* by the `let` expression in its *body* e'. That is, we may use the variable x within e' to refer to the variable introduced by the `let` expression. The *name* of the bound variable is not important. We could equally well have chosen the variable y as the variable x in the `let` expression, and used y instead of x in its body to refer to that variable. This is called *renaming of bound variables*, or, for historical reasons, *α-conversion*.

Variables that are not bound in an expression are *free*. For example, in the expression `let` x`=10 in` x+y the variable x is bound (by the `let`) in the expression x+y, and the variable y is free in that same expression. Every variable must be bound *somewhere* in a program, but when examining a subexpression in isolation we may encounter variables without explicit binding occurrences.

Determining where a given variable is bound in a program is called *scope resolution*. It is not entirely trivial because we may re-use bound variables. For example, consider the expression

```
let x=10 in (let x=11 in x+x)+x
```

Both the outer and the inner `let` expressions bind the variable `x`. The inner `let` binds *all* occurrences of `x` within its body; the outer `let` also binds all occurrences of `x` within its body, *except* those that occur within another binding of `x`. This, speaking informally, the value of the above expression would be $32 = (11 + 11) + 10$.

The general rule is this. A non-binding occurrence of a variable is bound at the *nearest enclosing* binder for that variable determined by moving upwards towards the root of the abstract syntax tree in which it occurs. If no binder for that variable is encountered, the variable is free within that phrase. Since a binder for a variable can be enclosed within the scope of another binder for the same variable, the inner occurrence is said to *shadow* the outer variable, or to *puncture the scope* of the outer binder.

Shadowing can always be avoided by renaming bound variables. For example, we can rewrite the next `let` expression given above as follows without changing its meaning, yet removing any doubt about which variable occurrences refer to which binder.

```
let x=10 in (let y=11 in y+y)+x
```

By calling the inner variable `y`, we avoid creating a hole in the scope of the outermost `let` expression.

Bound variables may be thought of as "generalized pronouns". In natural languages (such as English) we use "he", "she", and "it" to refer back to an entity that was previously introduced (bound) in the context of the utterance. We may regard these pronouns as three distinct bound variables that we use to refer to entities "bound" to these variables earlier in the context of the sentence.

Exercise 2
Define a function $FV(e)$ yielding the set of free variables *in expression e. The definition will be by induction on the structure of e, and will require an auxiliary function $FV(d)$ to compute the free variables of a declaration d.*

While it is handy to assume that distinct bound variables have distinct names, it is an undue burden on the programmer to insist on this rule. After all, suppose that you are linking with a million-line library: how would you know what variable names to avoid, and why on earth should you care? However, since the names of bound variables are not semantically significant, we can accrue the benefits of the "unique name" convention by adopting simple convention — we *identify* (equate) all phrases that differ only in the names of bound variables. For example, we regard the two version of the nested-`let` expression to be *identical*.

According to this convention pieces of abstract syntax are *equivalence classes* of ast's under the equivalence relation given by relating any two expressions that differ only in the names of their bound variables. When writing down an equivalence class, we must choose a representative of it. In doing so we make the tacit assumption that we choose representatives so that distinct bound variables have distinct names and, morever, so that the names are chosen to be apart from any given finite set of names. Thus would we choose the second of the two nested-`let` expressions as representative of the equivalence class. Moreover, if we wish, we can choose another representative in which, say, `x` does not occur at all. To satisfy this condition, we might choose this representative instead:

`let` z`=10 in (let` y`=11 in` y`+`y`)+`z

The convention of identifying expressions that differ only in the names of bound variables (*i.e.*, are α-convertible to one another) is called *higher-order abstract syntax (HOAS)*, in contrast to our earlier notion of *first-order abstract syntax (FOAS)*. FOAS exposes the hierarchical structure of an expression; HOAS exposes also its binding structure. Note, however, that for HOAS to be applicable, we must specify which expression constructors bind which variables in which sub-expressions. For example, we must specify that in the expression `let` x`=`e `in` e', we specified above that x is bound by the `let` within its body e'.

6 deBruijn Indices

The passage from FOAS to HOAS can be regarded as passing from tree representations to graph representations. An *expression graph* is a directed graph derived from the first-order abstract syntax tree by replacing occurrences of bound variables by an edge (or "pointer") leading to its binding occurrence. This makes clear the binding structure of the expression without having to resort to arbitrary naming conventions for variables. Each α-equivalence class determines a unique expression graph that we can regard as the canonical representation of that expression.

Expression graphs are not very "user-friendly" because they lack an obvious linear representation, and would be difficult to write on a page or manipulate with standard text editors. But for the purposes of abstract syntax they are superior, precisely because they abstract away from niceties such as choice of variable names.

How might an expression graph be represented in ML? The fundamental problem is that bound variable occurrences are "back pointers" that refer higher up in an abstract syntax tree. This is not directly representable using ML data type declarations; some other means must be introduced for referring back to earlier binding occurrences of variables.

One, rather heavy-handed, approach would be to use reference cells to represent general graph structure (say, via adjacency lists) and regard expression graphs as ordinary graphs. This is overkill, however, because there really is no need to admit mutation of an expression graph, other than as a means to create backward references. Judicious use of data abstraction would hide the representation details, but at the expense of pattern matching, which would no longer be possible.

Another approach, invented by the Dutch mathematican N. G. deBruijn, is the technique of *deBruijn indices*. Bound variables are not named; rather, each occurrence is replaced by an index determining the binding occurrence of that variable. The index is the number of nested binders within the scope of the binder to which the index refers.

An example will illustrate the idea. The expression

```
let x=10 in let y=11 in x+y
```

is represented in deBruijn notation by

let 10 in let 11 in $\boxed{2}$+$\boxed{1}$

Here $\boxed{i}$ is a reference to the variable bound by ith enclosing `let` expression, starting from 1.

A peculiarity of deBruijn indices is that the same variable may be referred to by several different indices, depending on the nesting depth of the occurrence. For example, the expression

```
let x=10 in (let y=11 in y+y)+x
```

is represented in deBruijn notation by the expression

let 10 in (let 11 in $\boxed{1}$+$\boxed{1}$)+$\boxed{1}$.

All bound variable references have index 1, but the third refers to the outermost binder, whereas the first two refer to the innermost. This is a direct consequence of the convention of counting outwards to the bind occurrence.

DeBruijn indices are often used in implementations of languages, because they eliminate the need to worry about α-conversion. They are tricky to handle correctly, in part because of the context-sensitivity of indices. If you're not careful, a reference to one variable can silently become a reference to another, creating havoc for the programmer.

7 Substitution

It frequently occurs that we will wish to substitute an expression e' for all free occurrences of a given variable x in another expression e, written $[e'/x]e$. The key to the proper definition of substitution is to avoid *capture* of free variables in e' by binders within e. For example, suppose that e is the expression

```
let y=10 in x+y
```

and e' is the expression

```
2*y+1
```

The result of substituting e' for free occurrences of x in e is *not*

```
let y=10 in (2*y+1)+y
```

but rather

```
let z=10 in (2*y+1)+z
```

Proper substitution requires that we rename `y` in e before substituting to avoid capturing free occurrences of `y` in e'. The idea is that the occurrence of `y` in e' is a *different* variable from the one introduced by the `let` occurring in e. By renaming the bound variable in e before substituting, we can avoid confusion.

The formal definition of substitution must take account of capture avoidance. We can exploit the higher-order abstract syntax convention by choosing representatives appropriately. Specifically, whenever we wish to perform the substitution $[e'/x]e$, we tacitly choose a representative of the equivalence class of e so that no free variable in e' is bound within e. This ensures that capture cannot occur. With this in mind, we can define substitution formally by induction on the structure of e as follows:

$$\begin{array}{rcll}
[e'/x]x & = & e' & \\
[e'/x]y & = & y & (x \neq y) \\
[e'/x]n & = & n & \\
[e'/x](e_1\texttt{+}e_2) & = & [e'/x]e_1\texttt{+}[e'/x]e_2 & \\
[e'/x](e_1\texttt{*}e_2) & = & [e'/x]e_1\texttt{*}[e'/x]e_2 & \\
[e'/x](\texttt{let}\,d\,\texttt{in}\,e) & = & \texttt{let}\,[e'/x]d\,\texttt{in}\,[e'/x]e & \\
[e'/x](y\texttt{=}e) & = & y\texttt{=}[e'/x]e & (y \notin FV(e')) \\
[e'/x](d_1\,\texttt{and}\,d_2) & = & [e'/x]d_1\,\texttt{and}\,[e'/x]d_2 &
\end{array}$$

Notice that we may tacitly assume that no binder governs a free variable in e' in this definition.