Converting Derivation Trees to Abstract Syntax Trees

When we parse a sentence using a grammar to produce a tree, that tree is called a derivation tree. Derivation trees show precisely how we came to understand the structure of a sentence. However, for many uses, we don't need all of the information provided in a derivation. Indeed, sometimes the amount of detail in a derivation tree makes it difficult to see important structure.

For this reason, we often use an alternative tree form when trying to understand a structure. An abstract syntax tree, or AST, gives us the essential structure of a parsed structure. Students often have difficulty understanding the notion of ASTs at first, because the rules for converting a derivation tree to an AST vary from language to language. Nevertheless, an AST always has the following properties:

- All of the interior nodes of an AST are operations.
- All of the leaf nodes of an AST are data.

Consider the lambda calculus expression $\lambda a.(ab)c$. Its derivation tree and corresponding AST are shown side-by-side.

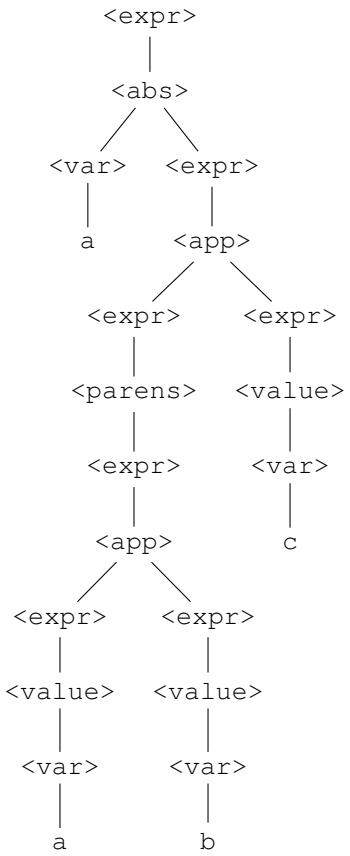

Figure 1: Derivation of $\lambda a.(ab)c$.

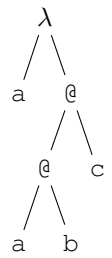

Figure 2: Abstract syntax tree for $\lambda a.(ab)c$.

Lambda Calculus Conversion Rules

In Figure 1 we have a complete record of how we determined the structure of a sentence using the class lambda grammar. By contrast, Figure 2 throws away a great deal of the derivation information, leaving us instead with a small tree that shows only operations and data. In the lambda calculus, the only operations are abstraction (λ) and application (`@`). Everything else is data.

Many students discover, with practice, that they can derive an AST directly from a lambda expression. If you have trouble seeing how this might be done, start by producing a derivation tree, then try converting the tree into an AST using the rules below. When you are done, discard the topmost `<expr>`.

Figure 3: Variables, where α is a variable like `x`.

Figure 4: Numbers, where v is a number like `1`.

Figure 5: Parentheses, where e is some expression.

Figure 6: Application, where e_1 and e_2 are expressions.

Figure 7: Abstraction, where α is a variable like `x` and e is an expression.