# Parsing: Hype for Sorts

Hype for Types: Lecture 3

Jeanne Luning Prak

February 2018

# 1 Syntax

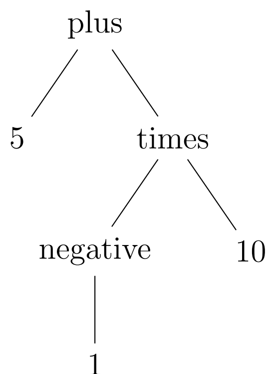
Today, we take a step back from talking about types to talk about a more basic kind of classification. Types give us information about the semantic meaning of expressions: what kind of value they will evaluate to if they do terminate, whether they can be properly executed by the language runtime, whether they are safe to use, etc. Types, generally, are specifications of program behavior. They give the programmer the guarantee "when my program runs, X will happen."[1]

However, types are not the only way of classifying pieces of programming languages. Before any typechecking can be done, language syntax must first be classified in another way: by sort. **Sorts** are a purely syntactic classification. Rather than making guarantees about the semantics of the language, they make guarantees about the structure of the language. To understand how sorts work and what they classify, let's first talk about the basics of programming language structure.

When you write code in your favorite language[2], all you're actually doing is typing characters into a text editor. Your program is literally one giant string. However, as you write it, you understand that you're not producing just a string - you're producing functions, variable declarations, while loops, etc. It's up to the **parser** of the language to take the string that you've written down and turn it into a format that represents the actual structure of the program.

This format is called an **abstract syntax tree**, or AST. An abstract syntax tree is a tree consisting of nodes, which can represent two things. One is **operators**, which are the building blocks of a language's syntax. These are the things you think of as you write the string that is your code: functions, while loops, variable declarations, arithmetic operations like plus, etc. The other is **variables**. Variables are simply placeholders in the AST, waiting for some other AST to be substituted in for them.

For example, the following is an AST:




which represents the expression `5 + (-1) * 10`. It can also be written in a more flat format as:

```
plus(5, times(negative(1), 10))
```

[1] Or at least, they should. Many languages lack this strong a guarantee from their type system, which makes us all a n g e r y.

[2] Which I assume is SML, OCaml, or Haskell.

ASTs are classified by sort. Sorts are broad categories of syntax. Some examples are given below:

| Sort | Example |
|---|---|
| Exp (expression) | `2 + 2` |
| Typ (type) | `(int * int) list` |
| Stmt (statement) | `x += 10;` |

Operators in ASTs are classified by **arity**, which describes the sorts of their subtrees and the sort of the tree that they are the root of. For example, we'd write the arity of plus as

```
(Exp, Exp) Exp
```

This means that plus has two subtrees, both of which have sort Exp, and the tree that it is the root of also has sort Exp.

*Exercise:* What is the sort of the AST drawn on the previous page? Assume that times has sort (Exp, Exp) Exp as well and that negative has sort (Exp) Exp.

Once we know the arity of each operator, we know what all syntax in the language must look like. For example, I know that a subtree of plus must never be a type, and if someone happened to type

```
x + int
```

into their text editor, they would get a parse error.

# 2 A Simple Language

Before we get into parsing, let's define the concrete syntax (the syntax you type into your text editor) and AST for a small language that we'll use as an example.

We'll informally define the concrete syntax and then give a more formal definition that's seen in the literature[3]

Our language will have:

1. Booleans: `true` and `false`
2. If expressions: `if e then e1 else e2` where e1 and e2 are expressions.
3. Value declarations: `val (x :  typ) = e;` where e is an expression, typ is a type, and x is a variable.
4. Variables: `x` where x is any string of numbers or letters of non-zero length that does not begin with a number.
5. Types: `bool`

We don't need to know what any of these pieces of syntax actually do, though the names are indicative of what we probably want them to do: if expressions case on the Boolean they're given, value declarations create variable bindings, and variables stand for previous bindings in value declarations.

More formally, we can define the same grammar using a syntax chart:

| | |
|---|---|
| Exp e ::= | `true` |
| | `false` |
| | `if e then e else e` |
| | `x` |
| Typ t ::= | `bool` |
| Decl d ::= | `val (x :  t) = e` |

Read from left to right, the chart reads as follows: On the left hand side are the sorts: Exp, Typ, and Decl. Then, a variable representing that sort that can be used in the syntax. Then finally, the syntax itself.

A program in this language might look like this:

```
val (x : bool) = true;
val (y : bool) = if x then false else true;
val (z : bool) = y;
```

From this, it's fairly simple to define an AST for this language. We take the sorts and make them into datatypes in our favorite language[4].

[3] This does not, in fact, just mean "In Bob Harper's book."

[4] SML

```
type variable = string

datatype exp = True
             | False
             | If of exp * exp * exp

datatype typ = Bool

datatype decl = Val of variable * typ * exp

type program = decl list
```

Now, what we want to do is take in a string representing our program and return a value of type program.

# 3 Parsing

This section is purely about the optional section of the homework. We didn't get to it in lecture, but if you're interested on learning how to implement a parser, feel free to read on.

Parsing is the process of taking a string and turning it into an AST. Because ASTs are trees, parsing is an inherently recursive process. In order to parse an if expression, I need to parse its three sub-expressions. However, the objects that parsers deal with are strings, which are not inherently recursive. Parsers must then make clever arrangements to discover recursive structure within a string. Often, they do this by using reserved words and operators such as if, =, and val to determine where they'll need to look for subtrees. A natural way to do this is to have a bunch of parsers that call other parsers when they think they'll find subtrees and then combine the results into the AST they're looking for:

```
decl_parser ("val x : bool = true;") "I see a val! Call the variable parser!"
variable_parser " : bool = true;"    "I see an x! I'll return it."
decl_parser (": bool = true;")       "I see a :! Call the type parser!"
type_parser "bool = true;"           "I see a bool! I'll return it."
decl_parser "= true;"                "I see an =! Call the exp parser!"
exp_parser ("true")                  "I see a true. I'll return it".
decl_parser                          "I've been given an x, a bool, and a true.
                                     I'll combine them into Val ("x", Bool, True)"
```

This is exactly what parser combinators do.

Combinators are higher-order functions that take functions as arguments, combine them together in some way, and spit out a function as the result. You already know of at least one combinator: function composition. $fog$ combines together two functions into a new function that first applies g to its argument and then applies f to the result of that.

Parser combinators are combinators that work on parsers. They take in two parsers and combine them together to create a new parser. Eventually, after applying many many parser combinators, you get one function that parses the whole program. Or at least that's the goal!

However, there's a tricky point here: parser combinators are functions that take in parsers and return parsers. They cannot interact with the input and output of the parsers they take in - only the parsers themselves (just like the composition operator doesn't get to know what the input to f and g will be). This turns out to be quite annoying. For example, I might want to combine two parsers in the following way: apply parser p1 to the input string, then save the result of p1, then apply parser p2 to the input string, then combine together the results of parsers p1 and p2 into an AST. This is incredibly common, since most ASTs have children, and so constructing them requires first parsing their children.

Parser combinators have a way of dealing with the potential output of a parser: two combinators called bind and with. Bind, denoted `--` in SML's Parcom library, takes in a parser and a function from the result of the first parser to some second parser, and returns a parser that first applies the first parser, then applies the second parser (with the result of the first parser in scope).

```
val p3 = p1 -- (fn x => p2) (* p3 applies p1,
```

```
                                then binds its result to x,
                                then calls p2 *)

val p3 = p1 wth (fn x => f x) (* p3 applies p1,
                                 then binds its result to x,
                                 then calls f x *)
```

Let's go through some examples of this. Let's try to parse an if expressions. To do this, we'll need a parser that parses expressions and a parser that parses the reserved operators `if`, `then`, `else`. Let's assume we have a parser that parses expressions for now. Call it exp.

```
val if_parser : exp parser =     reserved "if"
                              >>  exp
                              --  (fn b => reserved "then"
                              >>  exp
                              --  (fn e1 => reserved "else"
                              >>  exp
                              wth (fn e2 => If (b, e1, e2))
```

Here, `>>` is the same as bind, but it throws out the result of the first parser.

Let's do another example with a val declaration. Assume we have parsers for expressions and types.

```
val decl_parser : decl parser =    reserved "val"
                                >> identifier
                                -- (fn x => reservedOp ":"
                                >> typ
                                -- (fn t => reservedOp "="
                                >> exp
                                wth (fn e => Val (x, t, e))
```

In this week's code, there's a full example of how to parse the example language described above. Here's we'll just put a list of common, helpful combinators from SML's Parcom library:

```
parens p            [consumes a set of matched parens applies parser p inside them]
$p                  [used for mutually recursive parsers. The lazy
                     equivalent of p ()]
p1 <|> p2           [Attempts to apply p1 and if it fails, applies p2.
                     Any input consumed by p1 remains consumed.]
try p               [Attempt to apply p. If it succeeds, consume input.
                     Otherwise, fail and consume no input.]
p -- (fn x => p2)   [Applies p, then binds the result of p to x and
                     continues with p2]
p >> p2             [Same as --, but throws out the result of p]
```

```
p << p2             [Same as >>, but returns the result of p2 rather than p]
chainl1 p f         [Parses p n many times (n > 0) and applies foldl f
                     to the resulting list]
wth p f             [Parses p then applies f to the result]
```