COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 2: Grammars and Parsers

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

January 25, 2022

Review

Last time we compared and contrasted the two languages you may use for your projects in this class.

Today, we'll dive into programming language syntax and grammars and look at some of the issues that can come up when parsing programs.

Grammars

Programming language grammars are similar to human language grammars. Both give rules for forming valid "phrases" in the language. In the case of human language, those phrases may be whole sentences/subjects/predicates etc.

In the case of in the case of programming languages, those phrases may be whole programs/statements/expressions.

Grammar

Obviously, programming language grammars are simpler than human language. This is because they are context-free, while human language is not.

A language is context-free only if it is recognized by some push-down automaton. Language and automata theory has significant overlap with programming langauge theory, but we're primarily focused on implementation in this class.

Grammar

Programming language grammars are specified using **Backus-Naur Form**, BNF for short. A BNF grammar consists of four elements:

- A set of *terminals*, or tokens. Tokens are the smallest unit of syntax and form the vocabulary of the language. These are typically keywords or literal values like `while`, or `1.234`.
- A set of *non-terminals*, which represent abstract notions of syntactic structures in the language.
- A set of *production rules*, which map the non-terminals in the language to their expansions/patterns.
- A start symbol, which is a special non-terminal that represents the "goal" of the grammar.

These items are made more concrete if we take a look at an example.

Lambda Calculus Grammar

Let's make a syntax for the (untyped) lambda calculus. The lambda calculus is made up of three types of expressions:

- Variables, which we'll say are Haskell-style identifiers in our language. E.g. `x` or `take10`.
- Functions, a.k.a lambdas, a.k.a abstractions. We'll say these have the syntax of Haskell-style lambdas. E.g. `\x -> x`, `\x -> \y -> x`.
- Function application, which we'll represent using Haskell-style juxtaposition. E.g. `x y`, `(\x -> x) y`.

Another valid expression would be any of those expressions enclosed in parentheses (as implied by the last example above). Let's put these structures into a BNF grammar.

Lambda Calculus Grammar

A good place to start is to specify the terminals. What are the smallest units of the language? Let's look at some expressions we expect to be able to write in the language and think:

Lambda Calculus Expressions

```
fn
\x -> fn y
(\x -> x) fn
```

Lambda Calculus Grammar

Our terminals appear to be the set:

```
{ \ , -> , ( , ) } ++ Ident
```

Where `Ident` is the set of identifiers (and `++` is union). We now have the set of terminals; the smallest units of syntax in our language. That's a good start, now what about the non-terminals? What are the abstract syntactic structures we want to have in our language?

Lambda Calculus Grammar

When developing the non-terminals of a language, it helps to think hierarchically about the grammar, starting with the start symbol. In the case of lambda calculus, looking at it this way can feel weird. What is our start symbol, and what is a *program* in the lambda calculus?

The lambda calculus is slightly atypical in that a program can just be any of the expressions we mentioned above: Variables, functions, function applications, or parenthesized expressions. So we can get away with having a single non-terminal for expressions (programs), which will also be our start symbol!

Lambda Calculus Grammar

With our terminals, non-terminals, and start symbol in hand, we can finally formulate a set of production rules. The meta-syntax for doing this can vary, but it generally looks something like the following:

$$
\begin{aligned}
\langle exp \rangle \quad ::= \quad & \text{ident} \\
| \quad & \text{`\textbackslash'} \ \text{ident} \ \text{`->'} \ \langle exp \rangle \\
| \quad & \langle exp \rangle \ \langle exp \rangle \\
| \quad & \text{`('} \ \langle exp \rangle \ \text{`)'}
\end{aligned}
$$

Where ident is an arbitrary identifier.

These production rules show how the terminals and non-terminals are combined to form expressions/programs in the lambda calculus, and they specify basically everything we've said about the language up to now. Note that the non-terminals are enclosed in angle brackets.

Parsing

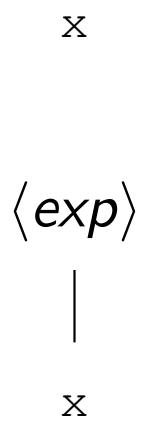
We now have a suitable specification of the language's grammar which we can use to actually try to parse expressions. If our specification was a real executable parser, we could simply try the parser on some expressions.

Instead, our specification is not executable, so we have to manually parse expressions. One way to do this is to build *parse trees* that illustrate the parsing of expressions. If the source text is a syntactically valid expression, then it is possible to derive a parse tree for the expression.

Parse Trees

The leaves of a parse tree are always terminals in the language. The internal nodes are always non-terminals whose children are the terminals/non-terminals on the right-hand side of a production.

For example, let's build a parse tree of the program `x`:



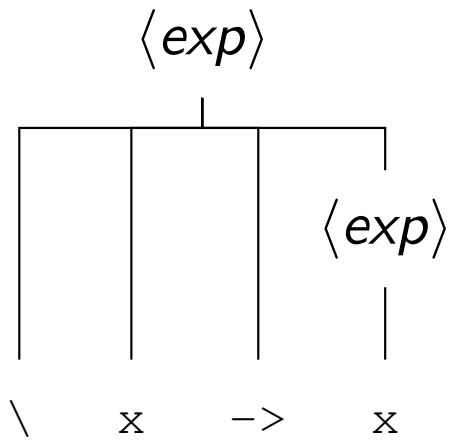
Parse Trees

That one was easy. What about this one: `\x -> x`

Parse Trees

That one was easy. What about this one: `\x -> x`

`\x -> x`



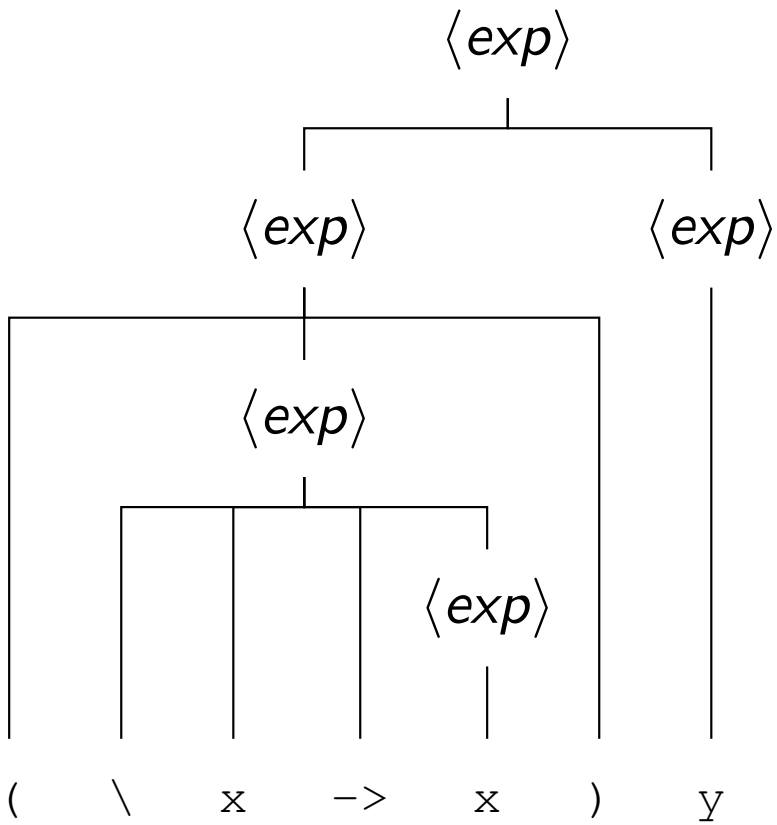
Parse Trees

But what about this one: `(\x -> x) y`

Parse Trees

But what about this one: `(\x -> x) y`

`(\x -> x) y`

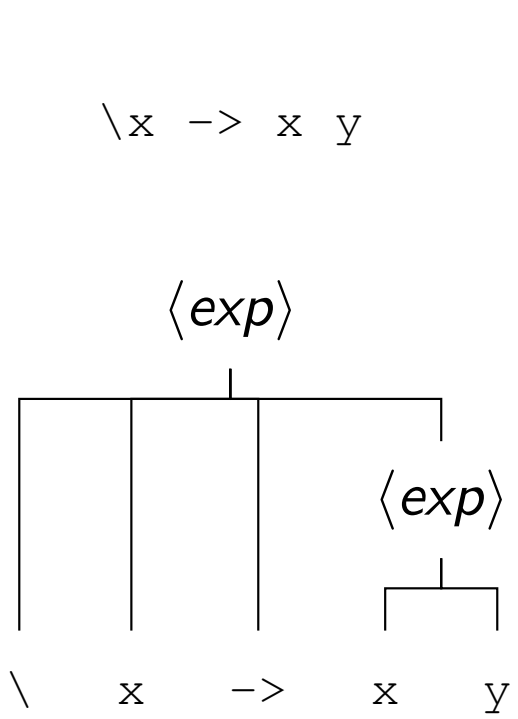


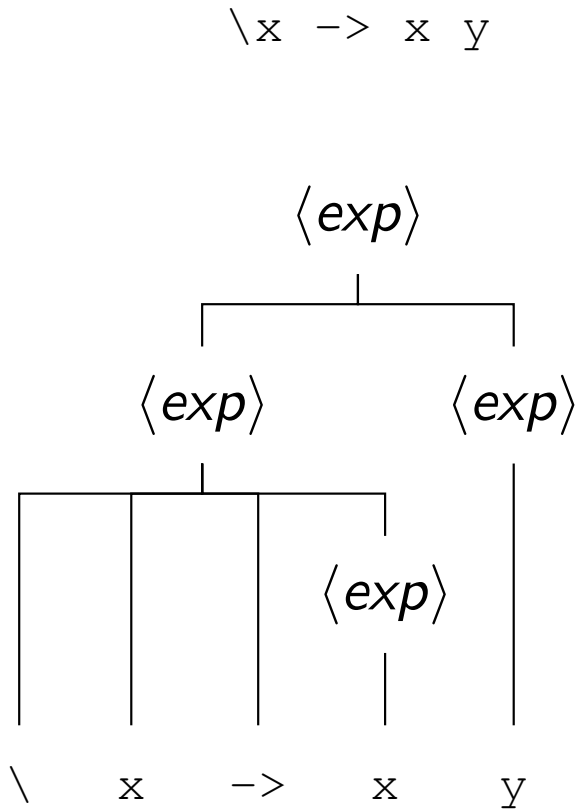
Parse Trees

It appears our grammar recognizes all of the programs that we want it to. However, consider this program: `\x -> x y`. How might we parse this?

Parse Trees

It appears our grammar recognizes all of the programs that we want it to. However, consider this program: `\x -> x y`. How might we parse this? There's two ways to parse this!





Ambiguity

The fact that the expression `\x -> x y` can be derived by two different parse trees means precisely that our grammar is *ambiguous*.

Conversely, if a grammar is ambiguous in any way, then there exists some source expression for which there is more than one parse tree. We'll talk about where ambiguity comes from and how to fix it. But first, what does ambiguity really mean in practice?

Ambiguity

If all we're concerned with is specifying a syntax for some language, it's often completely acceptable to specify the syntax ambiguously and explain away the ambiguity later. Or, if we're only concerned with preformed terms that aren't ambiguous *by construction*, we may not even address it.

Ambiguity

If all we're concerned with is specifying a syntax for some language, it's often completely acceptable to specify the syntax ambiguously and explain away the ambiguity later. Or, if we're only concerned with preformed terms that aren't ambiguous *by construction*, we may not even address it.

For example, the grammar Standford's programming language course uses for the lambda calculus:

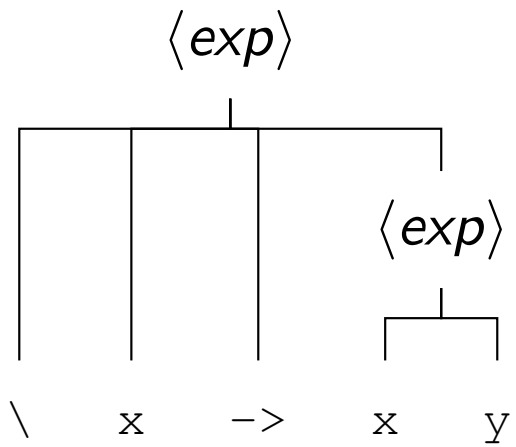
Expression e	$::=$	x	variable
	$\mid$	$\lambda\, x\, .\, e$	function definition
	$\mid$	$e_1\ e_2$	function application

But they jump straight into semantics, no parsing, so ambiguity be-gone!

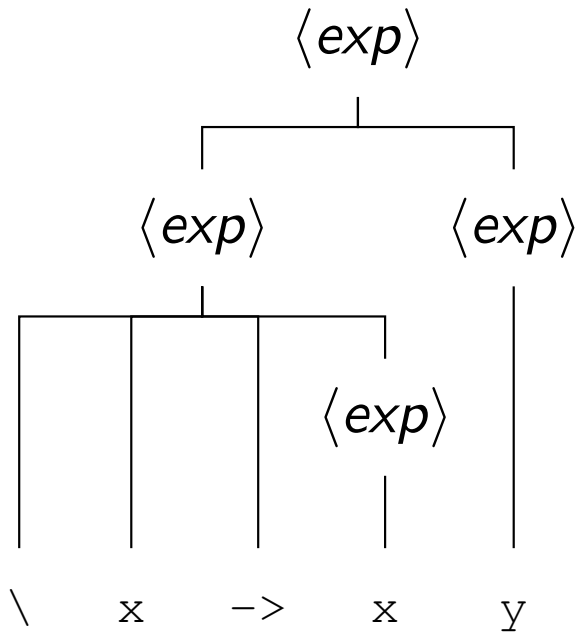
Ambiguity

On the other hand, if we're specifying a grammar that will actually be used for parsing in some manner, ambiguity can be bad. It forces the parser (whether human or machine) to make a decision about which parse tree is "correct". From our example above, which is correct?

```
\x -> x y
```




```
\x -> x y
```



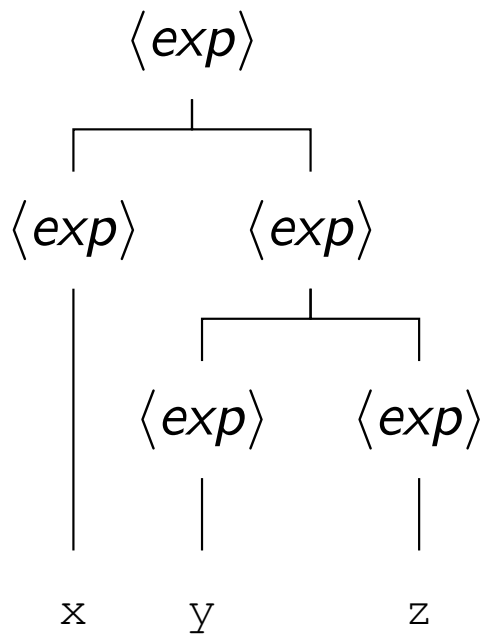
Ambiguity

We have intuition that tells us the tree on the left is the correct one. As another example, how might we parse the function application: `x y z`

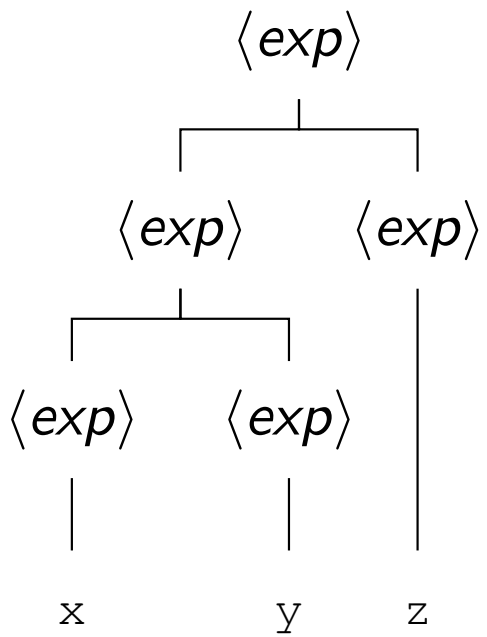
Ambiguity

We have intuition that tells us the tree on the left is the correct one. As another example, how might we parse the function application: `x y z`

`x y z`



`x y z`



Ambiguity

Again, there are two valid parse trees. If we use standard conventions, the tree on the right is "correct", since function application should associate to the left.

The conventions/intuitions I'm referring to have names, and they are our secret weapons when it comes to resolving ambiguity: *Precedence* and *associativity*.

Ambiguity

Specifically, the ambiguities in our lambda calculus grammar come from the fact that lambda abstractions and function application have the same precedence, and we haven't specified the associativity of the function application operation.

The question now becomes, how do we build these precedence and associativity rules into our grammar?

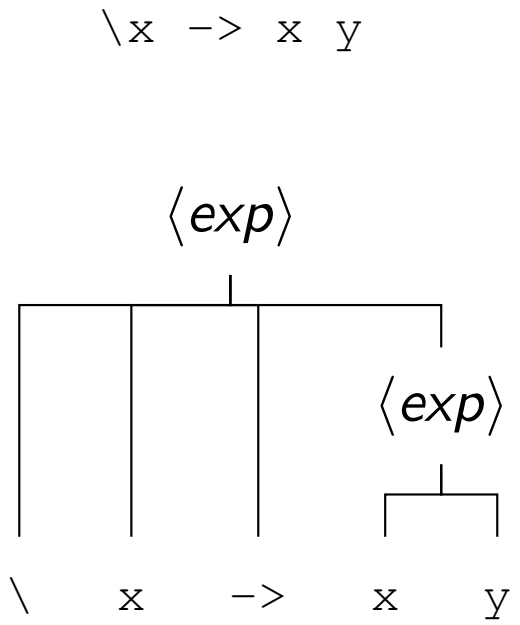
Precedence

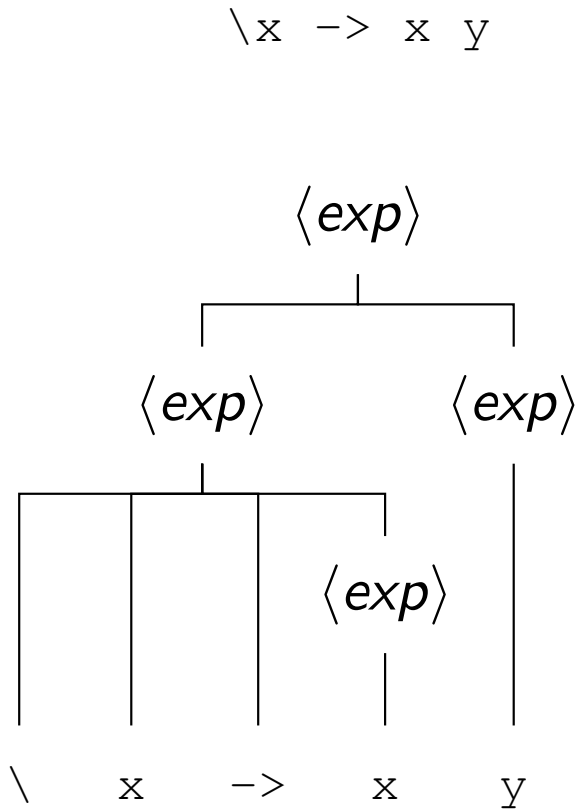
Once we have the precedence rules built in, associativity is trivial, so it's always good to focus on precedence. We want function application to have *higher precedence* than lambda abstraction.

In general, if some non-terminal in the language has higher precedence than another, than that non-terminal will always occur *further away* from the start symbol, meaning it will always be more deeply nested in the parse tree.

Precedence

Take a look at the parse trees (correct on the left) from our example earlier, and try to observe this "farther away" idea.





Precedence

Looking at our ambiguous grammar, can you think of any other precedence rules we may want to enforce?

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \text{ident} \\
& | & \text{`\textbackslash'} \ \text{ident} \ \text{`->'} \ \langle exp \rangle \\
& | & \langle exp \rangle \ \langle exp \rangle \\
& | & \text{`(')} \ \langle exp \rangle \ \text{`)'}
\end{array}
$$

Precedence

The precedence rules:

- Identifiers and parenthesized expressions should be allowed to occur as sub-expressions of any expression (highest precedence).
- Function applications should be allowed to occur as sub-expressions of other function applications and lambda abstractions.
- Lambda abstractions should only be allowed to occur as sub expressions of other lambda abstractions (lowest precedence).

Precedence

Those rules are now rather straight forward to translate into a BNF grammar, starting with the lowest precedence:

$$\begin{array}{lcl} \langle lam \rangle & ::= & \text{`\textbackslash'} \text{ ident } \text{`->'} \langle lam \rangle \\ & | & \langle app \rangle \end{array}$$

$$\begin{array}{lcl} \langle app \rangle & ::= & \langle app \rangle \langle app \rangle \\ & | & \langle var \rangle \end{array}$$

$$\begin{array}{lcl} \langle var \rangle & ::= & \text{ident} \\ & | & \text{`('} \langle lam \rangle \text{`)'} \end{array}$$

Precedence

All we've done is create non-terminal for every precedence level. It may be more clear if we name the non-terminals with their precedence levels. This also makes it clear that higher-precedence expressions may occur as lower-precedence expressions, but not vice versa.

$$
\begin{aligned}
\langle L0 \rangle \quad &::= \text{`\textbackslash'} \ \text{ident} \ \text{`->'} \ \langle L0 \rangle \\
&\mid \quad \langle L1 \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle L1 \rangle \quad &::= \langle L1 \rangle \ \langle L1 \rangle \\
&\mid \quad \langle L2 \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle L2 \rangle \quad &::= \text{ident} \\
&\mid \quad \text{`('} \ \langle L0 \rangle \ \text{`)'}
\end{aligned}
$$

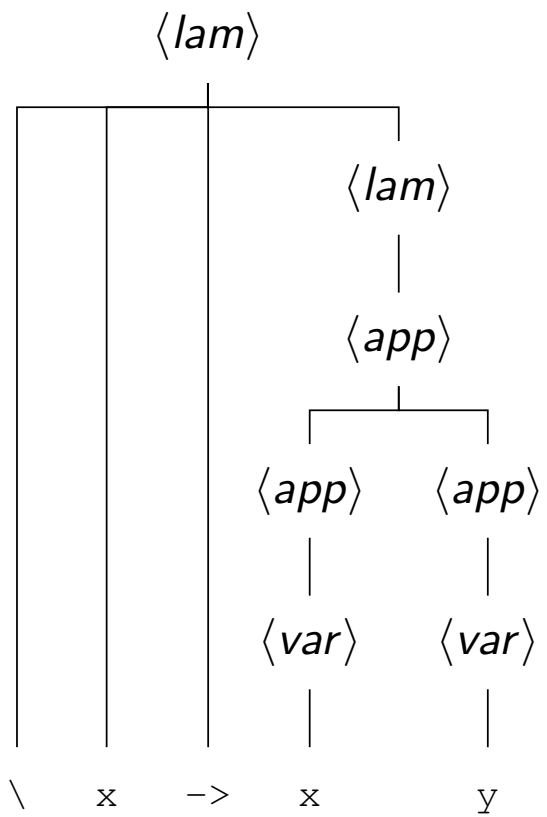
Precedence

Now what happens if we try to parse `\x -> x y`?

Precedence

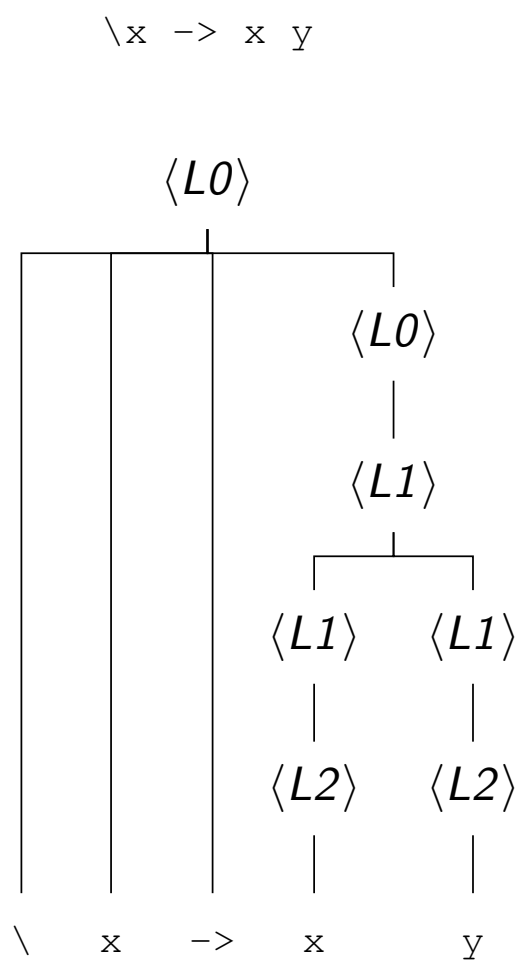
Now what happens if we try to parse `\x -> x y`?

`\x -> x y`



Precedence

With the precedence labels for non-terminals:

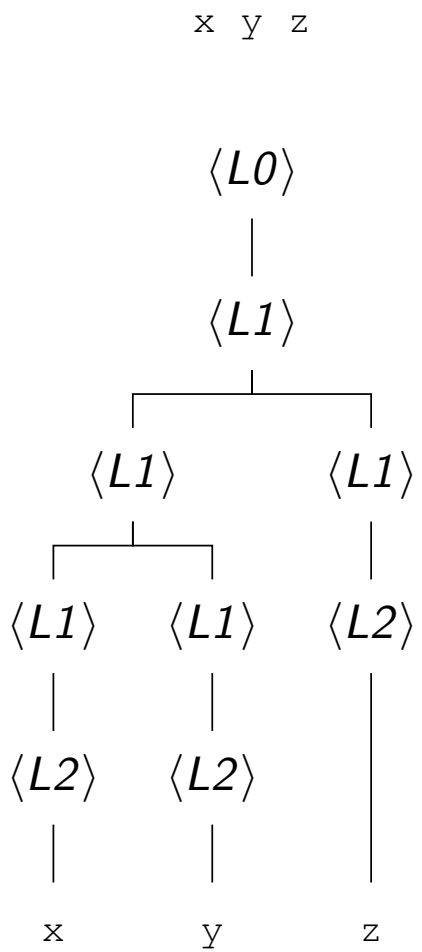


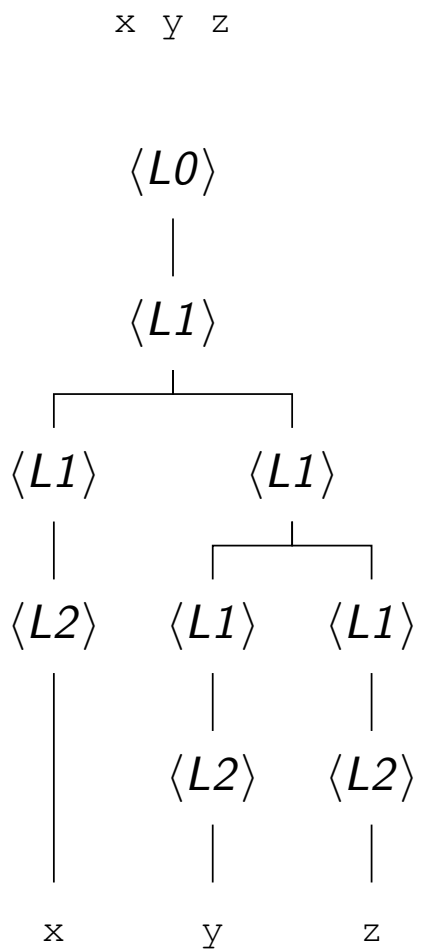
Associativity

Now that we have the precedence rules sorted, have we fixed associativity? Try parsing `x y z`:

Associativity

Now that we have the precedence rules sorted, have we fixed associativity? Try parsing `x y z`: Still ambiguous!





Associativity

If we follow conventions, we want function application to be left-associative, meaning `x y z` should be the same as `(x y) z`. How can we add this to our new grammar?

$$
\begin{aligned}
\langle L0 \rangle \quad &::= \text{`\textbackslash'} \ \text{ident} \ \text{`->'} \ \langle L0 \rangle \\
&\mid \quad \langle L1 \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle L1 \rangle \quad &::= \langle L1 \rangle \ \langle L1 \rangle \\
&\mid \quad \langle L2 \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle L2 \rangle \quad &::= \text{ident} \\
&\mid \quad \text{`('} \ \langle L0 \rangle \ \text{`)'}
\end{aligned}
$$

Associativity

We just want to make sure a function application cannot occur as the right hand expression inside another function application! Change the grammar like this:

$$
\begin{aligned}
\langle L0 \rangle &::= \text{`\textbackslash'} \ \text{ident} \ \text{`->'} \ \langle L0 \rangle \\
&\quad | \quad \langle L1 \rangle \\
\\
\langle L1 \rangle &::= \langle L1 \rangle \ \langle L2 \rangle \\
&\quad | \quad \langle L2 \rangle \\
\\
\langle L2 \rangle &::= \text{ident} \\
&\quad | \quad \text{`('} \ \langle L0 \rangle \ \text{`)'}
\end{aligned}
$$

Associativity

We just want to make sure a function application cannot occur as the right hand expression inside another function application. Change the grammar like this:

$$
\begin{aligned}
\langle L0 \rangle &::= \text{`\textbackslash'} \ \text{ident} \ \text{`->'} \ \langle L0 \rangle \\
&\mid \langle L1 \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle L1 \rangle &::= \langle L1 \rangle \ \langle L2 \rangle \\
&\mid \langle L2 \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle L2 \rangle &::= \text{ident} \\
&\mid \text{`('} \ \langle L0 \rangle \ \text{`)'}
\end{aligned}
$$

Associativity

We call rules like the following *left-recursive*, since the first item on the right hand side is the non-terminal of the rule itself:

$$\begin{aligned} \langle L1 \rangle \quad &::= \langle L1 \rangle \ \langle L2 \rangle \\ &\mid \quad \langle L2 \rangle \end{aligned}$$

In general, left-recursive rules will create left-associativity, and right-recursive rules will create right-associativity, e.g.:

$$\begin{aligned} \langle L1 \rangle \quad &::= \langle L2 \rangle \ \langle L1 \rangle \\ &\mid \quad \langle L2 \rangle \end{aligned}$$

Takeaways

- Programming language grammars can be specified using BNF grammars.
- Those grammars may be ambiguous, which allows some expressions to be parsed in more than one way.
- Sometimes ambiguity is okay. If the grammar is being used to specify a parser, ambiguity is sketchy since it leaves decisions to the parser.
- Ambiguity can be eliminated by building precedence and associativity rules into the grammar itself.
- Precedence determines which non-terminals may occur where. It can also be though of as specifying "parse order".
- Associativity is simple once we have precedence rules.

Next Time

Next time, we'll put away the parse trees (for good) and see how BNFC can help us generate parsers and iterate on grammars quickly. We'll basically redo everything we did in this lecture, but largely automated, and we'll see how the ambiguities are handled by the executable parsers.

Useful Resources

- Wikibook for programming language grammars
- Handy notes on precedence/associativity

The End