CSCI 334:
Principles of Programming Languages

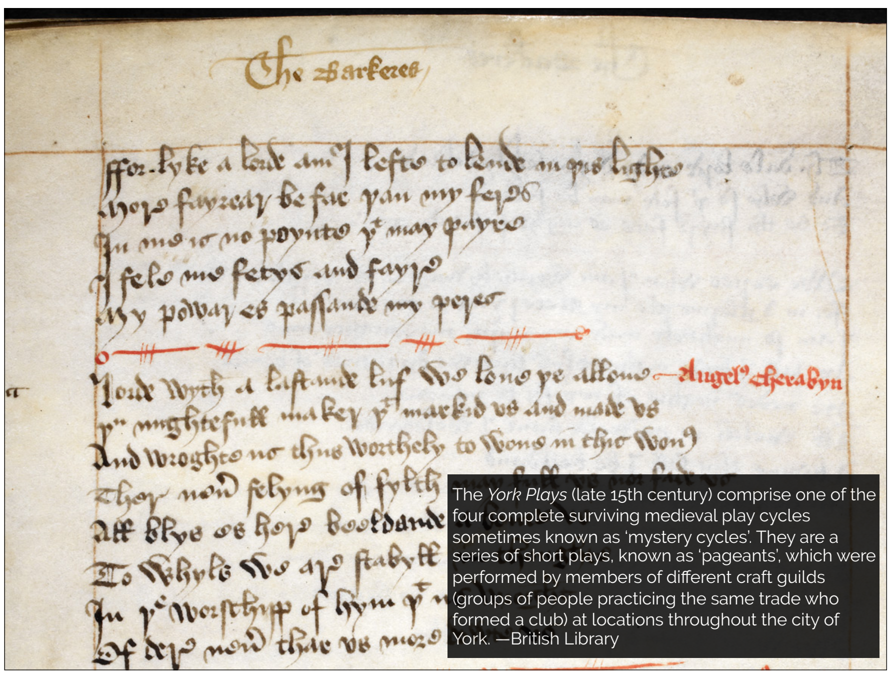
Lecture 4: PL Fundamentals II

Instructor: Dan Barowy

Williams

Announcements

- How did Lab 2 go?
- Lab 3 posted (pset)
- Small errors in book figures (thanks, Edwin!)



The *York Plays* (late 15th century) comprise one of the four complete surviving medieval play cycles sometimes known as 'mystery cycles'. They are a series of short plays, known as 'pageants', which were performed by members of different craft guilds (groups of people practicing the same trade who formed a club) at locations throughout the city of York. —British Library

Why couldn't you understand the script?

It's written in English, after all!

We don't know the "ground rules" for the document as it is written:

- Surface appearance ("syntax")
  - What is the set of valid symbols?
  - What combinations of symbols are permissible?
- Deeper meaning ("semantics")
  - How does a given arrangement of symbols correspond to meaning?

Formal language

A **formal language** is the set of permissible **sentences** whose **symbols** are taken from an **alphabet** and whose word **order** is determined by a specific set of **rules**.

Intuition: a language that can be defined mathematically, using a **grammar**.

English **is not** a formal language.

Java **is** a formal language.

More formally

$\mathcal{L}(\mathbf{G})$ is the set of all sentences (a "language") defined by the grammar, **G**.

G = (**N**, **Σ**, **P**, **S**) where

N is a set of nonterminal symbols.

Σ is a set of terminal symbols.

P is a set of production rules of the form
N ::= (**Σ**∪**N**)*
where * means "zero or more" (Kleene star) and
where ∪ means set union

S∈**N** denotes the "start symbol."

Backus-Naur Form (BNF)

More concretely, for programming languages, we conventionally write **G** in a form called BNF.

Nonterminals, **N**, are in brackets: `<expression>`

Terminals, **Σ**, are "bare": `x`

A production rule, **P**, consists of the `::=` operator, a nonterminal on the left hand side, and a sequence of one or more symbols from **N** and **Σ** on the right hand side.

```
<variable> ::= x
```

The `|` symbol means "alternatively": `<num> ::= 1 | 2`

We use **ε** to denote the empty string nonterminal.

Backus-Naur Form (BNF)

You should read the following BNF expression:

```
<num> ::= <digit>
        | <num><digit>
```

as

"`num` is defined as a `digit` or as a `num` followed by a `digit`."

Backus-Naur Form (BNF)

The following definition should look familiar:

```
<expr>  ::= <num>
          |  <expr> + <expr>
          |  <expr> - <expr>
<num>   ::= <digit>
          |  <num><digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

`<expr>` is the start symbol.

Conventionally, we ignore whitespace, but if it matters, use the ␣ symbol. E.g.,

```
<expr>␣+␣<expr>
```

Lambda calculus grammar

```
<expr>  ::= <var>
          |  <abs>
          |  <app>
<var>   ::= x
<abs>   ::= λ<var>.<expr>
<app>   ::= <expr><expr>
```

`<expr>` is the start symbol.

Pro tip

Don't try to "understand" the lambda calculus.

Aside from "variables," "functions," and "application," it has no more meaning than regular algebra.

We ascribe meanings to it later (as we do with algebra).

The lambda calculus is simply a tool for reasoning by using the logic of computation.

Parse Trees

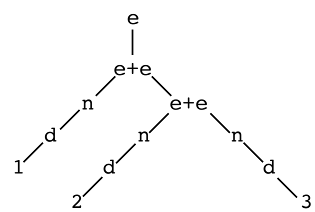
There are at least two forms of trees that we might refer to "parse trees"

Derivation Tree

Describes exactly how input was parsed

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

1+2+3

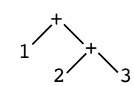


Abstract Syntax Tree

Abstracts over representation details

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

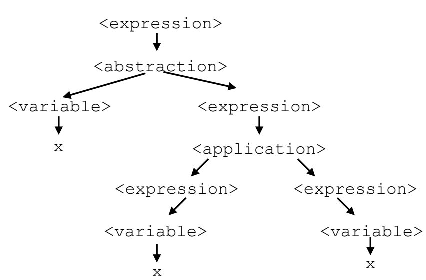
1+2+3



Parse tree

We can create a “parse tree” by following the rules of a grammar as we interpret a sentence of a language.

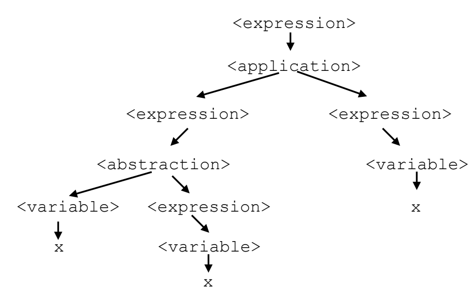
λx.xx



Abiguity

You might have noticed that there is an alternative parse tree.

λx.xx



Parentheses disambiguate grammar

```
<expr> = (<expr>)
```

Axiom of equivalence for parens

Let's modify our grammar

Lambda calculus grammar

```
<expr>   ::= <var>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= x
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
```

While we're at it…

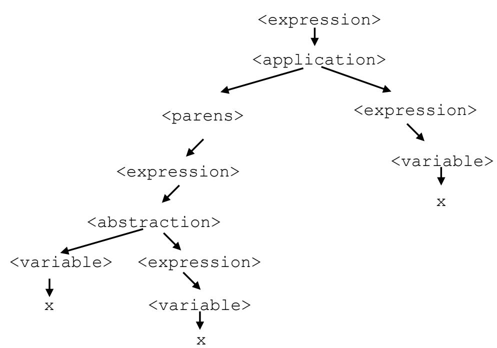
```
<expr>   ::= <var>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
```

Also…

```
<expr>   ::= <value>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
<value>  ::= v ∈ ℕ
           |  <var>
```

This expression is now unambiguous

(λx.x) x



Free vs bound variables

(λx.x) x

bound

free

Evaluation: Lambda calculus is like algebra

(λx.x) x

Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

α-reduction

β-reduction

α-Reduction

(λx.x) x

This expression has two *different* x variables

Which should we rename?

Rule:

λx.<expr> $=_\alpha$ λy.[y/x]<expr>

[y/x] means "substitute y for x in <expr>"

α-Reduction

`(λx.x) x`

`(λy.[y/x]x) x`

`(λy.y) x`

β-Reduction

`(λx.x) y`

How we "call" or *apply* a function to an argument

Rule:

`(λx.<expr>) y` $=_\beta$ `[y/x]<expr>`

Reduce this

`(λx.x) x`

How far do we go?

We keep going until there is nothing left to do

`x` ← done

`xx` ← done

`λx.y` ← done

`(λx.xy) z` ← not done

That "most simplified" expression is called a *normal form*.

An expression that can be simplified is a called a *redex*.

Sometimes multiple simplifications

Order (mostly) does not matter

$$\begin{tikzcd}[column sep=small] & M \arrow[dl] \arrow[dr] & \\ M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\ & N & \end{tikzcd}$$

If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

"confluence"

Example

```
(λa.λb.(- a b)) 2 1
```

Activity

Leftmost reduction:

```
(λf.λx.f(f x))(λz.(+ x z))2
```

Activity

Rightmost reduction:

```
(λf.λx.f(f x))(λz.(+ x z))2
```

Recap & Next Class

Today we covered:

Lambda calculus

Next class:

Lambda calculus

Computability