CSCI 334:
Principles of Programming Languages

Lecture 6: The Dream of Computation

Instructor: Dan Barowy
Williams

1

Topics

Syntax in Backus-Naur Form

What can computers really do?

Lambda calculus

2

Your to-dos

1. Read *Syntax*, **for Thursday 2/22**.
2. Lab 3, **due Monday 2/26** (solo lab)
   Give yourself enough time to learn a small amount of $\LaTeX$

3

Announcements

- CS Colloquium this **Friday, Feb 23 @ 2:35pm in Wege Auditorium (TCL 123)**

How Big is YouTube?
Prof. Ethan Zuckerman ('93, UMass Amherst)

Social media and user-generated content have thoroughly transformed the media landscape, giving birth to powerful companies, transforming news and political participation. Despite the influence of platforms run by Google and Meta, it is difficult to answer the simplest questions about these technologies, like "How many videos are hosted on YouTube?" Our lab has published a novel method for estimating the size of YouTube and learning other essential facts about the platform. In the process, we have uncovered a set of legal and ethical questions that will be essential for other "unpermissioned research" about social media platforms.

4

Class poll

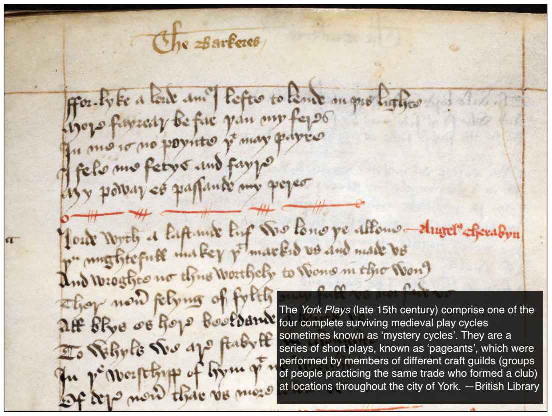
- Two suggestions
  - Extend deadline to midnight (OK)
  - Return quizzes before deadline (solutions instead)

5

Language of languages

6

Stepping up several levels, how do we talk about computer languages without talking about a *specific* computer language? How can we talk about them *generally*? It turns out that they all have some parts in common.



7

Please read this for me. You can't? Why not? It's written in English!

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:

- Appearance: **syntax**
  - What is the set of valid **symbols**?
  - What **arrangements** of symbols are permissible?
- Meaning: **semantics**
  - What does **a given arrangement** of symbols correspond **mean**?

8

Although this is technically "English," it is unlikely that you would be able to read it, as the rules of English have changed over time. For a language, there are essentially two families of rules.

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.

9

We've briefly discussed this before…

More formally

$L(G)$ is the set of all sentences (a "language") defined by the grammar, G.

$G = (N, \Sigma, P, S)$ 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
$N ::= (\Sigma \cup N)^*$
where * means "zero or more" (Kleene star) and
where $\cup$ means set union

$S \in N$ denotes the "start symbol."

10

What *really* is a formal language? What's the "form"? The form is G = (N, sigma, P, S).

Backus-Naur Form (BNF)

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

John Backus

Peter Naur

Invented in 1959 to describe the ALGOL 60 programming language.

11

Although you could define a language using pure set theory, we prefer a more convenient, but equivalent syntax: BNF. BNF was created to be able to describe the syntax of any programming language, but it was specifically developed when ALGOL was being designed.

Tower of Hanoi (ALGOL 60)

```
begin
  procedure movedisk(n, f, t);
  integer n, f, t;
  begin
    outstring (1, "Move disk from");
    outinteger(1, f);
    outstring (1, "to");
    outinteger(1, t);
    outstring (1, "\n");
  end;

  procedure dohanoi(n, f, t, u);
  integer n, f, t, u;
  begin
    if n < 2 then
      movedisk(1, f, t)
    else
      begin
        dohanoi(n - 1, f, u, t);
        movedisk(1, f, t);
        dohanoi(n - 1, u, t, f);
      end;
  end;

  dohanoi(4, 1, 2, 3);
  outstring(1,"Towers of Hanoi puzzle completed!")
end
```

12

ALGOL looks a lot like a modern programming language! In fact, many textbooks use ALGOL as a kind of algorithm pseudocode.

Backus-Naur Form (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.

13

How does BNF work? It works like this.

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`."

14

Here's a recursive snippet. Observe that this example allows us to describe numbers of any length. The definition is not complete, however, because it contains no terminals (but it would be OK if you defined <digit> in terms of terminals).

Backus-Naur Form (BNF)

The following definition might 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>
```

15

Here's a definition for a simple language that can add multi-digit numbers. This is a complete definition.

Parsing and Parse Trees

Parsing is the process of analyzing a string of symbols, conforming to the rules of a formal grammar, to understand:

1) whether that sentence is valid (**s** ∈ L(**G**)), or
2) the structure (e.g., “parts of speech”) of that sentence (a **parse tree**).

16

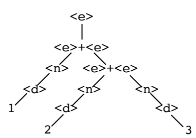
So how do we “interpret” sentences? First, we need to derive their structures using the rules of the grammar. This process is called “parsing.”

Derivation Tree

Shows **every step** of how a sentence is **parsed**.

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

```
1+2+3
```



17

Given a BNF grammar and an expression in that language, we can trace through the steps we use to recognize a valid expression. If you do that, you get what is called a “derivation tree.”

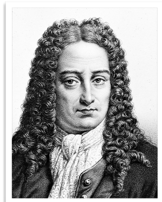
What can computers really do?

18

Now that we have a basic idea of how we might describe syntax—which is the subject of this week’s lab—we have some idea of how we might formulate a language for communicating with a computer. So now let’s discuss what we might talk to them about: what can and can’t computers do, fundamentally?

The Dream

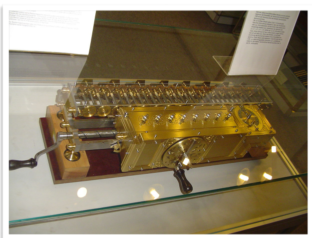
"I thought again about my early plan of a new language or writing-system of reason, which could serve as a communication tool for all different nations... If we had such an universal tool, we could discuss the problems of the metaphysical or the questions of ethics in the same way as the problems and questions of mathematics or geometry. That was my aim: Every misunderstanding should be nothing more than a miscalculation (...), easily corrected by the grammatical laws of that new language. Thus, in the case of a controversial discussion, two philosophers could sit down at a table and just calculating, like two mathematicians, they could say, 'Let us check it up …'"



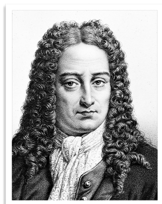
Wilhelm Gottfried Leibniz

19 One of the first people to wonder about this question was this bewigged dude, the genius and polymath, Leibniz.

The Dream

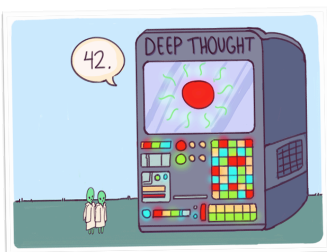


"stepped reckoner"



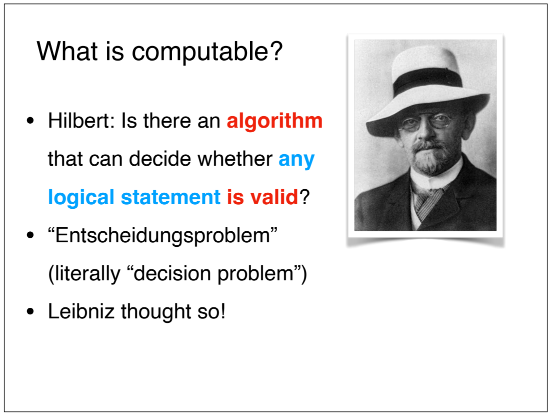
Wilhelm Gottfried Leibniz

20 Leibniz actually attempted to build machines that realized his dream. He started small, with a machine called the "stepped reckoner" that could perform arithmetic: addition, subtraction, multiplication, and division. This basic design was used (for real!) for more than 200 years!



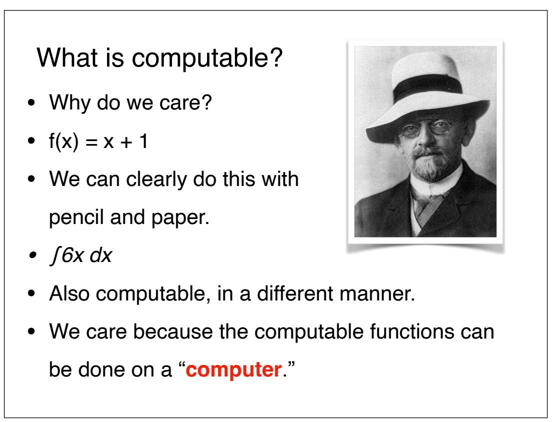
"What is the answer to the ultimate question of life, the universe, and everything?

21 Anyone here a fan of Douglas Adams? In the Hitchhiker's Guide to the Galaxy, philosophers build a machine to answer essentially the same question, but they phrase their question so imprecisely as to get nonsense out of the machine. "garbage in, garbage out" This, of course, was comedy, but anyone who has read Douglas Adams before knows that there's always something deep and interesting at the center of his jokes.



22

Over the years, though, what we mean when we ask “what can computers do?” has gotten more precise. It’s true that the stepped reckoner could only do arithmetic, but that does not mean that we could not build a more powerful machine. Hilbert thought this was a very important question: what’s the most powerful machine we could build? His idea was to separate the idea of what a specific machine could do from the idea of what machines could do in principle, and his was the first precise use of the work “algorithm.” Hilbert thought that the “toughest” problem was simply for a machine to state, for a given logical statement, whether the statement was true or false.



23

Many mathematical problems reduce to this formulation. For example, clearly arithmetic has a form that we can say true/false things about. And calculus does too, although the steps are maybe a little different. But we care, because at their heart, proving things about them is similar, and we can imagine that anything that is “computable” in this sense can be computed on a machine.

Lambda calculus

- Invented by Alonzo Church in order to solve the Entscheidungsproblem.
- Short answer to Hilbert's question: no.
- Proof: **No algorithm can decide equivalence** of two arbitrary λ-calculus expressions.
- By implication: no algorithm can determine whether an arbitrary logical statement is valid.

24

There's some interesting history here about Gödel that I am going to gloss over, but the first person to really take a stab at the problem in the way that Hilbert meant was this guy, Alonzo Church. To do that, he invented a little language that he thought captured everything important about computation. That language was called the lambda calculus. The lambda calculus is computational logic in its purest form. And Church showed that there is no algorithm that can decide the equivalence of two lambda calculus expressions. So in essence, no algorithm can determine whether an arbitrary logical statement is valid. This was obviously very disappointing to lots of people, but, as it turns out, the devil is in the details. We can still do a lot with computers!

What is the meaning of x in **algebra**?

25

Let's spend a little time investigating the lambda calculus. Remember: this is a different system of logic. Let's start simply by looking at something that is familiar to you. In algebra, what does x mean?

Pro tip

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

Aside from "**variable**," "**function definition**," and "**application**," it has no inherent meaning.

We **ascribe meaning** to it, just as we do with algebra.

The lambda calculus is simply a **system** for reasoning by using **the logic of functions**.

26

Many first-timers get hung up on details of the lambda calculus. Remember: there is no inherent meaning in a lambda calculus expression. What a given expression means depends on how you ascribe meaning to it, just as with algebra.

Lambda calculus grammar

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

`<expr>` is the start symbol.

27

Here is the syntax of the lambda calculus, expressed in BNF.

What is a variable?

```
<var>   ::= x
```

It's just a value.

28

So what is a variable? It's just a value. Which value? It does not really matter, in the same way that the value of x does not really matter in algebra.

What is an abstraction?

```
<abs>   ::= λ<var>.<expr>
```

It's a function definition

```
def foo(x):
  <expr>
```

29

What is abstraction? It's a function definition.

What is an application?

```
<app>   ::= <expr><expr>
```

It's a "function call"

```
foo(2)
        <expr><expr>
```

function   argument

30

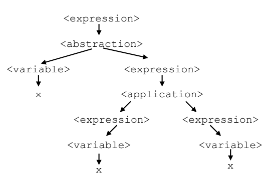
What is application? It's a function call.

That's it. That's all of the lambda calculus.

Parsing Lambda Expressions

Let's try parsing this expression

λx.xx



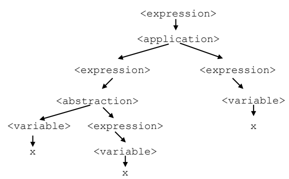
31

For now, though, let's focus on derivation. What is the derivation for this lambda calculus expression?

Ambiguity

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

λx.xx



32

Note that BNF does not always capture every necessary detail. For example, here is another potential derivation for the same expression. However, this derivation is not correct because the lambda calculus DOES include additional rules to eliminate ambiguity. These rules, called precedence and associativity, are the most difficult rules for newcomers.

Abiguity

In fact, the lambda calculus is never ambiguous because of its precedence and associativity rules—see the reading.

33

Parentheses disambiguate grammar

`<expr> = (<expr>)`

Axiom of equivalence for parens

Let's modify our grammar

34

Parens make precedence and associativity rules strictly unnecessary (assuming that parens have the highest precedence). We will keep the precedence and associativity rules for the lambda calculus, but if you want to use parens to help you understand an expression, feel free to insert them.

Lambda calculus grammar

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

35

While we're at it…

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

36

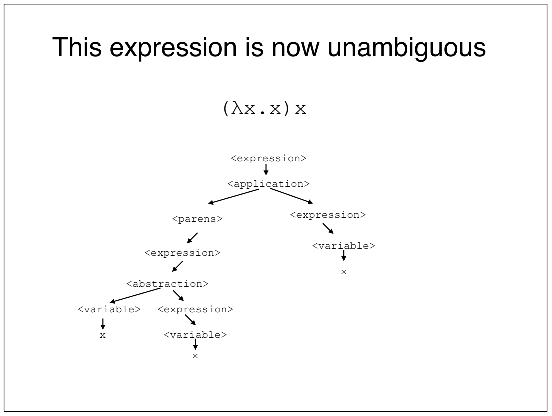
Also, it is very helpful to have variables other than x.

Also…

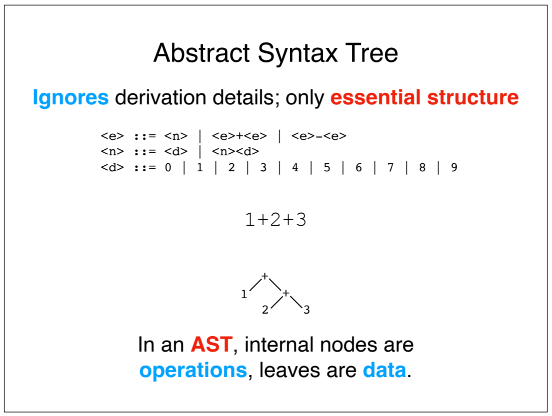
```
<expr>   ::= <value>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
<value>  ::= v ∈ ℕ
           |  <var>
```

37

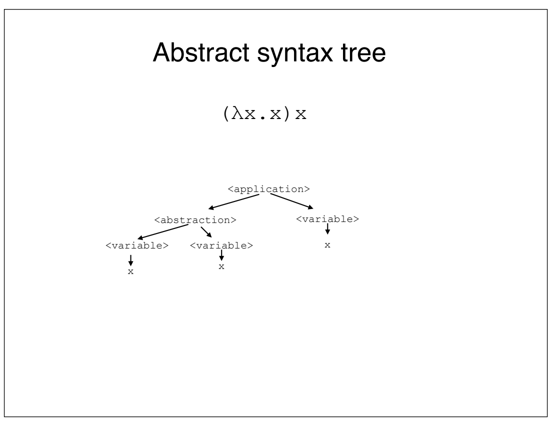
Finally, we will sometimes add arbitrary literal values to the lambda calculus. These are not strictly necessary, but they make working with the language a little easier.



38 With parens, our original expression is unambiguous. It's this parse.



39 Later on in this semester, we will ignore tiny details in derivation and focus on a more meaningful version of a parse tree, called an "abstract syntax tree." ASTs better get at what a given expression means.



40 Eventually, you will see that the abstract syntax tree tends to be more useful than derivation trees, so we often favor parse trees in this form.

Recap & Next Class

Today:

BNF

Lambda calculus / computation

Next class:

More on lambda calculus

41