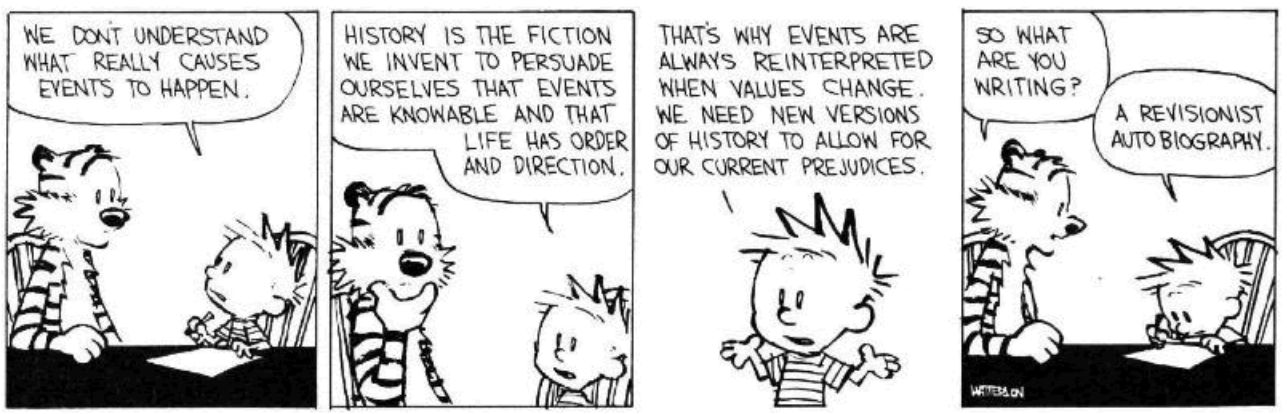
Meeting 08 - Abstract Syntax and Parsing

Bor-Yuh Evan Chang

Thursday, September 19, 2024

Meeting 08 - Abstract Syntax and Parsing



What questions does your neighbor have?

In-Class Slides

Book Chapter

Announcements

- HW 2 due this ~~Friday~~ Monday 9/23 6pm

Today

- Revisit last bit of Concrete Syntax
- Abstract Syntax
- Triage Your Questions
  - Using VSCode or the terminal to test your code (on coding.csel.io)?
  - Auto-testing with GitHub Actions?
  - Lab 1?
- Revisit and Go Deeper On:
  - Concrete Syntax (Meeting 07), if time permits

Questions?

- Review:
  - How do you show that a *grammar* is *ambiguous*?

give string + 2 parse trees (Ar.)

- How to show grammar is ambiguous?
- How do parsers deal w/ ~~order of~~ precedence ~~operations~~? (Andrew) (Alan)

PEMDAS

"Rules of Precedence"

highest → lowest

Questions?

One-Slide Review

Concrete syntax is …

Abstract syntax is …

Parsers convert …, which have deal with …

… and … are ways to to describe and deal with a common form of ambiguity.

An Ambiguous Grammar

$$\text{expressions} \quad e \quad ::= \quad n \mid e \,/\, e \mid e - e$$

A Related Unambiguous Grammar

$$
\begin{array}{rrcl}
\text{expressions } \texttt{Expr} & e & ::= & \texttt{N(}n\texttt{)} \\
 & & \mid & \texttt{Divide(}e_1\texttt{, }e_2\texttt{)} \\
 & & \mid & \texttt{Minus(}e_1\texttt{, }e_2\texttt{)} \\
\text{integers} & n & &
\end{array}
$$

Try hard to read this as concrete syntax. It is unambiguous, right? Why?

Abstract Syntax Trees

```
1  sealed trait Expr                                        // e ::=
2  case class N(n: Int) extends Expr                        //   n
3  case class Divide(e1: Expr, e2: Expr) extends Expr       // | e1 / e2
4  case class Minus(e1: Expr, e2: Expr) extends Expr        // | e1 - e2
```

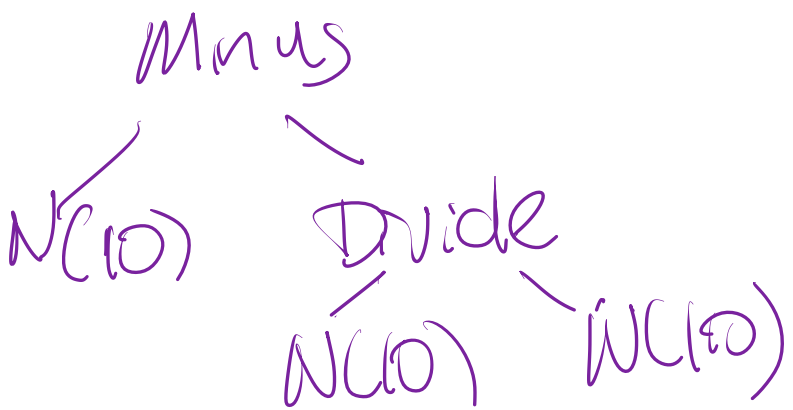
3+3

345

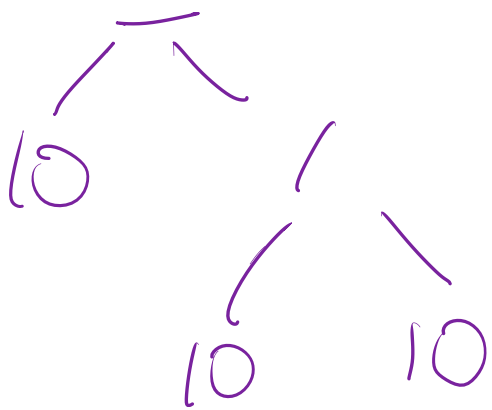
Abstract Syntax Trees

```
Minus(N(10), Divide(N(10), N(10)))
```

```
graph TD;
  A[Minus]
  B["N(10)"]
  C[Divide]
  D["N(10)"]
  E["N(10)"]

  A --> B
  A --> C
  C --> D
  C --> E
```

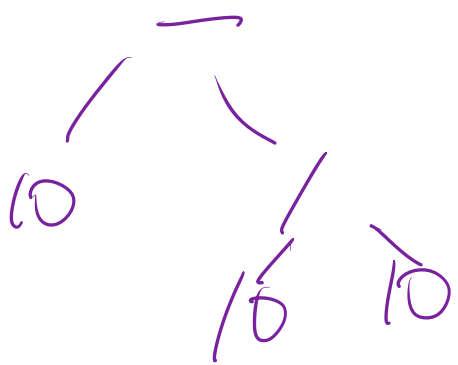



```
graph TD;
  A["-"]
  B["10"]
  C["/"]
  D["10"]
  E["10"]

  A --> B
  A --> C
  C --> D
  C --> E
```



Abstract Syntax Trees versus Parse Trees

```
graph TD;
  A["-"]
  B["10"]
  C["/"]
  D["10"]
  E["10"]

  A --> B
  A --> C
  C --> D
  C --> E
```



```
graph TD;
  A["e"]
  B["e"]
  C["-"]
  D["e"]
  E["n(10)"]
  F["e"]
  G["/"]
  H["e"]
  I["n(10)"]
  J["n(10)"]

  A --> B
  A --> C
  A --> D
  B --> E
  D --> F
  D --> G
```

Parsing

Take a Theory of Computation course for more language theory. We focus here on the practical aspect of reading and writing BNF grammars.

There are also many advanced parser libraries and generator tools. You might use one in a Compiler Construction course.

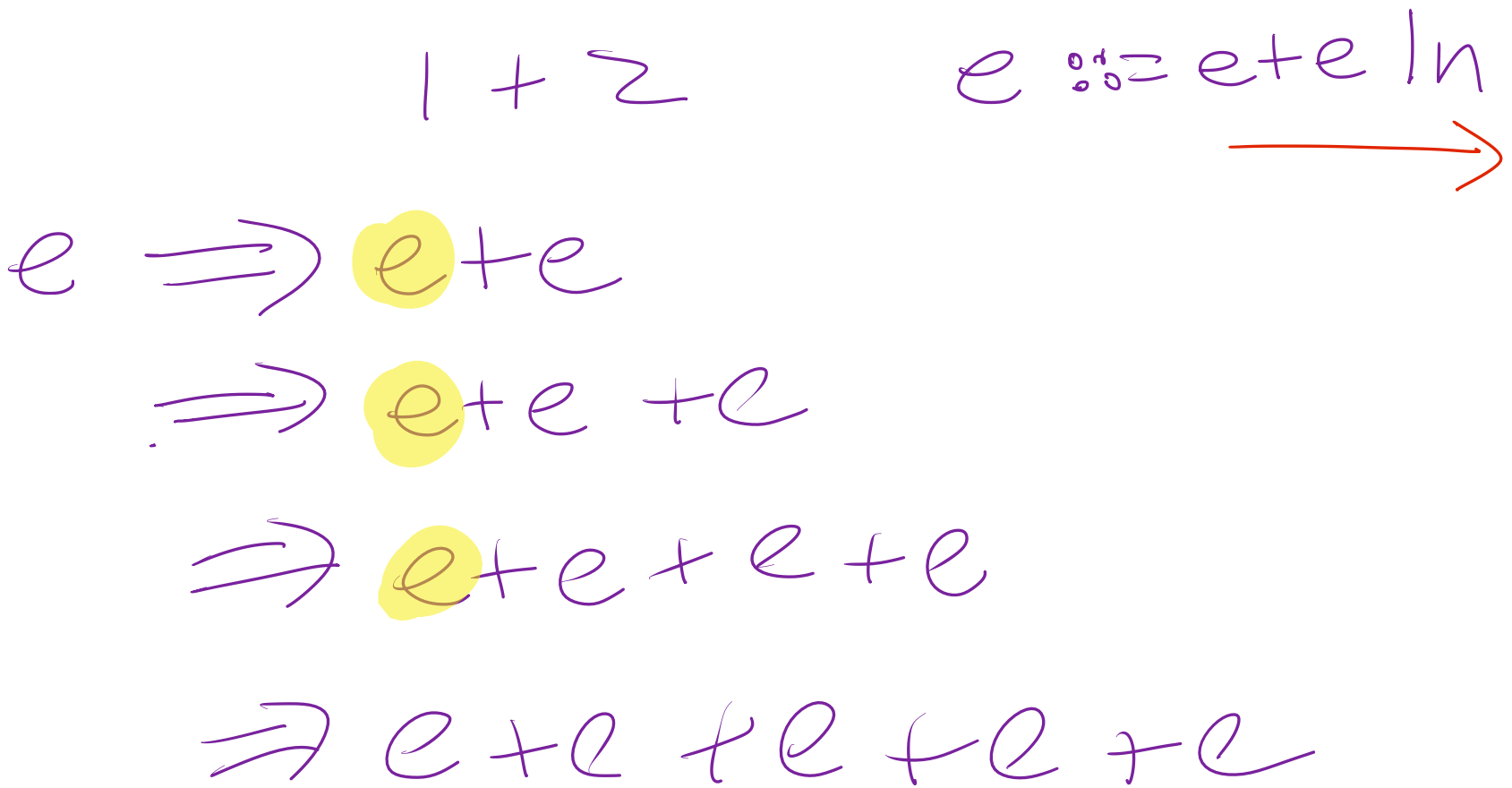
An Ambiguous Grammar

$$\begin{array}{rrcl} \text{expressions} & e & ::= & n \mid e + e \\ \text{numbers} & n & & \end{array}$$

```
defined trait Expr
defined class N
defined class Plus
```

Recursive-Descent Parsing

Automate the top-down, *leftmost parsing derivation*.



Recursive-Descent Parsing

Two rules:

1. unambiguous grammars
2. no left recursion in the grammar

Combinator Parsing

```
import $ivy.$
```

What's a *combinator*?

↳ higher-order function / method

For Lists, you used

for loop → foreach

Restricting the Concrete Syntax

term ::= num | (expr)

expr ::= term + term

Let’s Implement a Parser!

```
defined object ExprParser
```