CS4400/5400
Programming λanguages

Lecture 3
Interpreters & abstract syntax

Spring 2024

Instructor: Steven Holtzen

s.holtzen@northeastern.edu

Northeastern University
Khoury College of Computer and Information Sciences

Goals for today

1. Build calc, our very first language in Plait
2. Learn how to run an interpreter by hand
3. Understand abstract syntax
4. If time: parsing

Logistics

- Reminder: Homework #2 due Wednesday
- Next homework coming out Wednesday, due Friday Feb 2
- At this point, you should be quite comfortable programming in Plait
- Course webpage up (see Canvas syllabus page)
  - https://pages.github.khoury.northeastern.edu/sholtzen/cs4400-spr24/

What is our eventual goal?

- We are going to give the syntax and semantics for `calc`:

```
> (calc (parse `(+ 1 2)))
- Number
3
> (calc (parse `(+ 1 (+ 2 3))))
- Number
6
```

Recall: syntax and semantics

Syntax

What does a program look like?

Python

```
x = 5
print(x)
```

Semantics

What does a program do?

- Create a variable called “x”
- Print the contents of that variable

Recall: syntax and semantics

Syntax

What does a program look like?

Formal descriptions as grammars

Semantics

What does a program do?

Programs that run programs

Interpreters!

Syntax

The presentation of programs

Goal

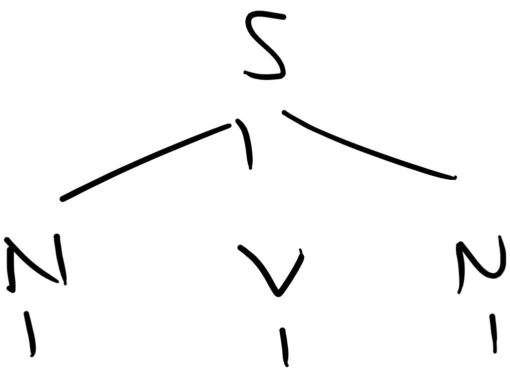
- Give the syntax for a tiny calculator programming language
  - Support numbers and addition
  - Be able to write programs like “1 + 2” and “3 + 4 + 10”

Parsing spoken language

- Sentences are formed by building complex phrases out of smaller ones
  - The rules for this process are called *grammars*
- A ***noun*** is an object
  - ”dog”, “Steven”, …
- A ***verb*** is an action
  - “eats”, “cuts”, …
- A ***tiny sentence*** consists of a noun (subject), a verb, and a noun (object)
  - “Steven eats food”

Parsing spoken language

- **Sentence parsing**: extracting the grammatical structure of a sentence from its presentation



Steven eats food

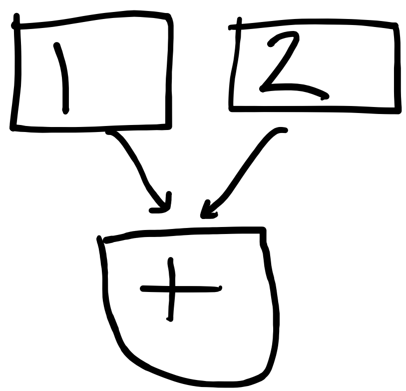
N. Chomsky
Phrase structure grammar

What is the syntax of programs?

- Syntax is the **presentation** of a program
  - What you give to the computer
  - Text, diagrams, etc.

`(+ 1 2)`

S-exp



Circuit

`1 + 2 + 3`

Infix

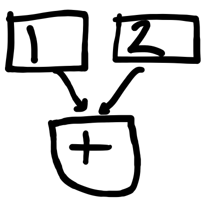
- There is wide variability in program syntax
  - People have different aesthetic preferences

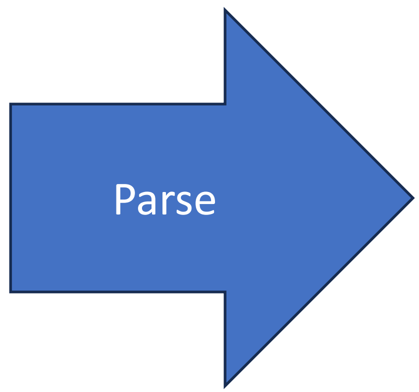
Abstract syntax

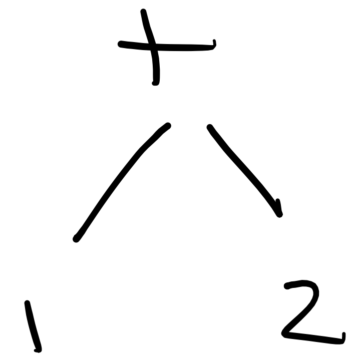
- First big idea: Abstract away the details of the program presentation

`(+ 1 2)`

`1 + 2`







Surface syntax

- Easy for programmers to write
- Concise and aesthetically pleasing
- Called “Surface” to distinguish it from “abstract”

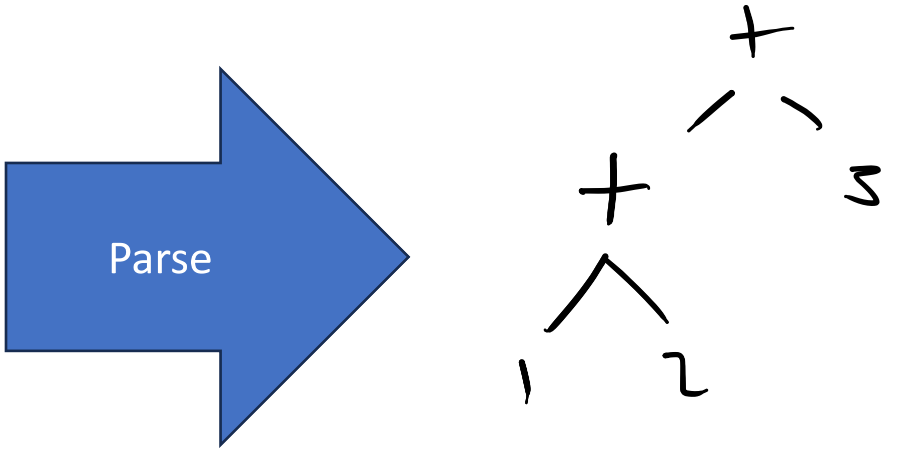
Abstract syntax

- Easy for computers to understand
- Precise and unambiguous

Abstract syntax tree (AST)

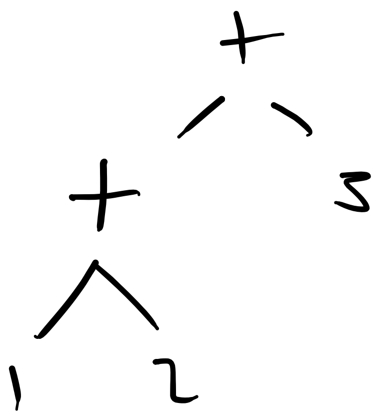
- An AST is a tree-based representation of surface-syntax
  - Each node in the tree is called a ***term***
  - The child of a term is a ***sub-term***
  - Translating surface syntax to abstract syntax is called ***parsing***

1 + 2 + 3



Representing ASTs in Plait

```
(define-type Exp
    [num (n : Number)]
    [plus (left : Exp) (right : Exp)])
```




```
> (plus (plus (num 1) (num 2)) (num 3))
- Exp
(plus (plus (num 1) (num 2)) (num 3))
```

Some exercises

- Build the `calc` AST for the expression (written with infix notation):

  ((1 + 2) + (3 + 4)) + 5

- Build the calc AST for the expression (written with s-expression notation):

  (+ (+ (+ 1 2) (+ 3 4)) 5)

Ponder

- We could have chosen an alternative datatype for calc:

```
(define-type Exp
    [num (n : Number)]
    [plus (left : (Listof Exp))])
```

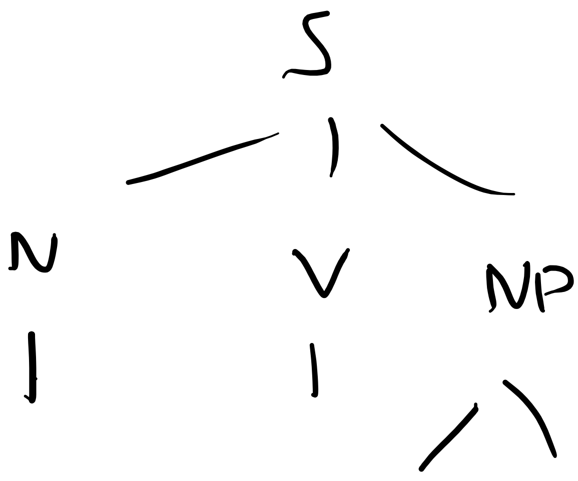
- What are the pros and cons of this choice of abstract syntax?

Semantics

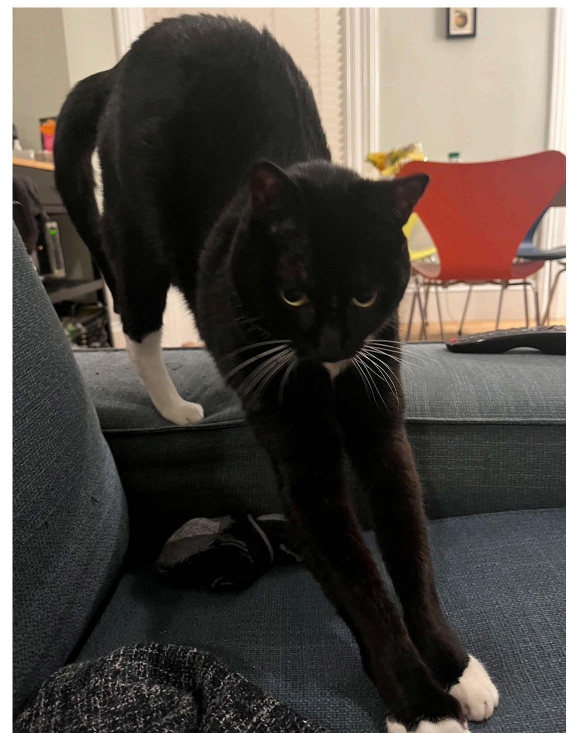
The meaning of programs

What is semantics?

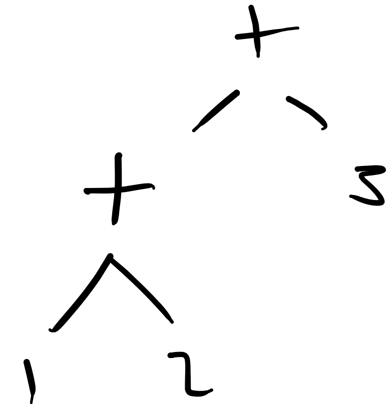
- Associate **syntactic** with a **meaning**

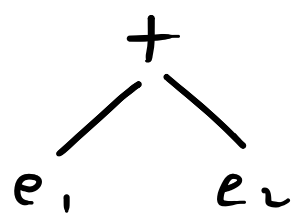


Steven has a cat



Semantics of `calc`



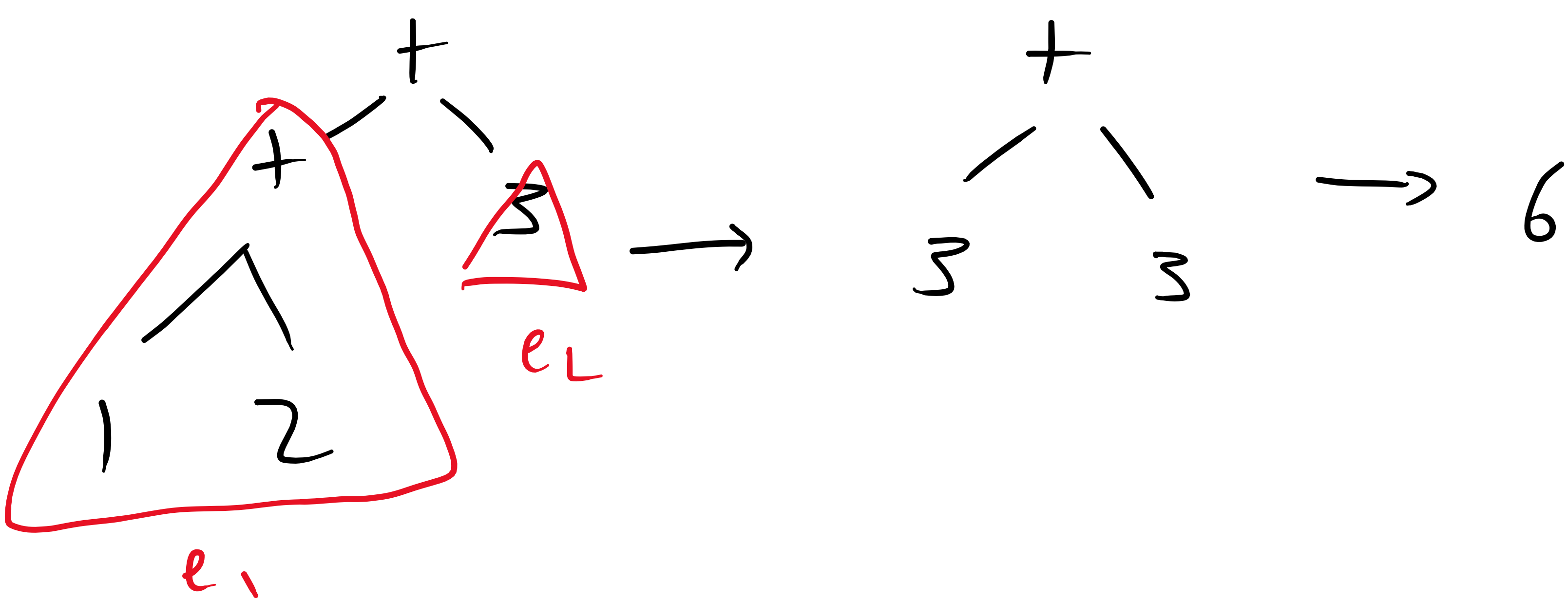
- Give a ***meaning*** to every ***term*** by describing what they ***evaluate** to*
- Numbers evaluate to numbers
- To evaluate 

, first evaluate e1, then e2,

  then evaluate their sum

Evaluating by hand

- To evaluate by hand, draw a sequence of ***steps***



"first evaluate e1"

Implementing an evaluator in Plait

```
(calc : (Exp -> Number))
(define (calc e)
  ???)
```

Start by writing the type and the shape of the function

Implementing an evaluator in Plait

```
(calc : (Exp -> Number))
(define (calc e)
  (type-case Exp e
    [(num n) ???]
    [(plus l r) ???]))
```

Following the design recipe, we can fill in the type-case to destruct the argument

Implementing an evaluator in Plait

```
(calc : (Exp -> Number))
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus e1 e2) ???]))
```

The base case is easy:
“Numbers evaluate to numbers”

Implementing an evaluator in Plait

```
(calc : (Exp -> Number))
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus e1 e2)
       (+ (calc e1) (calc e2))]))
```

The inductive case: first evaluate e1, then e2, then evaluate their sum

This program is called an ***interpreter for* `calc`**, and it gives the ***precise semantics of* `calc` *programs***

Interpreting (running) `calc` programs

```
> (calc (num 10))
- Number
10
> (calc (plus (num 1) (num 2)))
- Number
3
> (calc (plus (plus (num 1) (num 2)) (num 3)))
- Number
6
```

Syntax and semantics of `calc`

Abstract syntax

```
(define-type Exp
    [num (n : Number)]
    [plus (left : Exp)
          (right : Exp)])
```

Semantics

```
(calc : (Exp -> Number))
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus e1 e2)
       (+ (calc e1)
          (calc e2))]))
```

Ponder

- We could have chosen an alternative semantics for `calc` where we evaluate e2 before e1:

```
(calc : (Exp -> Number))
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus e1 e2)
       (+ (calc e2) (calc e1))]))
```

- Is this semantics fundamentally different from the one that evaluates e1 first? Why or why not?

Ponder

- Suppose we were using our list-based abstract syntax from earlier:

```
(define-type Exp
    [num (n : Number)]
    [plus (left : (Listof Exp))])
```

- What should the semantics of the plus with an empty list be? What should we do?
  - (there is no right answer here; there are pros and cons)

Parsing

From syntax to abstract syntax

Surface syntax

- We want to release our `calc` program to the world

- One option: make programmers simply give us `calc` ASTs

```
(plus (plus (num 1) (num 2)) (num 3))
```

- This is a bit undesirable; why do they need to tell us 1 is a number, and I would like to use “+” instead of “plus”

    - As our languages get more complex, such small annoyances become unbearable

Surface syntax

- What surface syntax should we choose for our `calc` language? We can choose many…
- Let's choose s-expressions!

```
(+ 1 (+ 2 3))
```

- Plait has very good built-in support for parsing and manipulating s-expressions

The s-expression datatype

- Every s-expression is either:
  - A constant (Boolean, empty list, number, symbol, string character)
  - A list of Plait s-expressions
- We could write this as a datatype:

```
(define-type S-Expr
  [bool (b : Boolean)]
  [empty]
  [num (n : Number)]
  [symbol (s : Symbol)]
  [string (s : String)]
  [list (l : (Listof S-Expr))])
```

Fun fact: Plait programs are s-expressions!

Constructing s-expressions

- S-expressions are a built-in Plait datatype
- Constructed using a backtick `

```
> `1
- S-Exp
`1
> `#t
- S-Exp
`#t
> `dog
- S-Exp
`dog
```

Constructing s-expressions

- The backtick *turns a Plait term into an s-expression*

```
> `(+ 1 2)
- S-Exp
`(+ 1 2)
> `(hello + world 123)
- S-Exp
`(hello + world 123)
> `(what (are you) doing)
- S-Exp
`(what (are you) doing)
> `(+ (+ 1 2) 3)
- S-Exp
`(+ (+ 1 2) 3)
```

We can also use curly braces `` `{+ 1 2} ``

Testing s-expressions

```
> (s-exp-number? `10)
- Boolean
#t

> (s-exp-number? `t)
- Boolean
#f

> (s-exp-boolean? `#t)
- Boolean
#t

> (s-exp-symbol? `hello)
- Boolean
#t

> (s-exp-list? `(1 2 3))
- Boolean
#t
```

Destructing s-expressions

```
> (s-exp->boolean `#t)
- Boolean
#t

> (s-exp->boolean `oops)
- Boolean
. . s-exp->boolean: not a boolean: `oops

> (s-exp->number `10)
- Number
10

> (s-exp->symbol 'hello)
. typecheck failed: S-Exp vs. Symbol in:
  s-exp->symbol
  (quote hello)

> (s-exp->symbol `hello)
- Symbol
'hello

> (s-exp->list `(1 2 3))
- (Listof S-Exp)
(list `1 `2 `3)
```

S-expressions as `calc` syntax

- We will use s-expressions to give a convenient surface syntax to `calc`

```
> (parse `(+ (+ 1 2) 3))
- Exp
(plus (plus (num 1) (num 2)) (num 3))
```

A parser for `calc`

- Start with its signature:

```
(parse : (S-Exp -> Exp))
(define (parse s)
  ???)
```

- Some tests:

```
(test (parse `1) (num 1))
(test (parse `{+ 1 2}) (plus (num 1) (num 2)))
(test/exn (parse `{1 + 2}) "")
```

A parser for `calc`

- Fill in the cases:

```
(parse : (S-Exp -> Exp))
(define (parse s)
  (cond
    [(s-exp-number? s)
     ???]
    [(s-exp-list? s)
     ???]
    [else (error 'parse "not recognized")]))
```