CS4400/5400
Programming λanguages

Lecture 4
Parsing and conditionals

Spring 2024

Instructor: Steven Holtzen

s.holtzen@northeastern.edu

Northeastern University
Khoury College of Computer and Information Sciences

Goals for today

1. Finish parsing and running `calc` programs
   - Host semantics
   - How to read grammars
   - Parsing s-expressions

2. Develop `cond`, extending calc with conditionals
   - See the design space of if-expressions
   - Create an evaluator for `cond`
   - The `value` datatype

Logistics

- New homework released tonight
  - Due next Wednesday (Jan 31), recommend you start
- Reminder: course materials on course webpage now instead of Canvas (see link on canvas)
  - All slides, code, etc. goes there
- I’ve been tweaking the schedule a bit based on our pace, check it for latest reading
- Reminder: I am fairly closely following PLAI, use it as a resource

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))]))
```

Host semantics

- I never actually told you what syntactic "+" did

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

- The semantics of "plus" in `calc` is inherited from Plait's semantics of "+"
- `calc` ***inherited the host semantics*** of "+" from Plait

Host semantics

- Q: What is Plait's semantics of "+"?

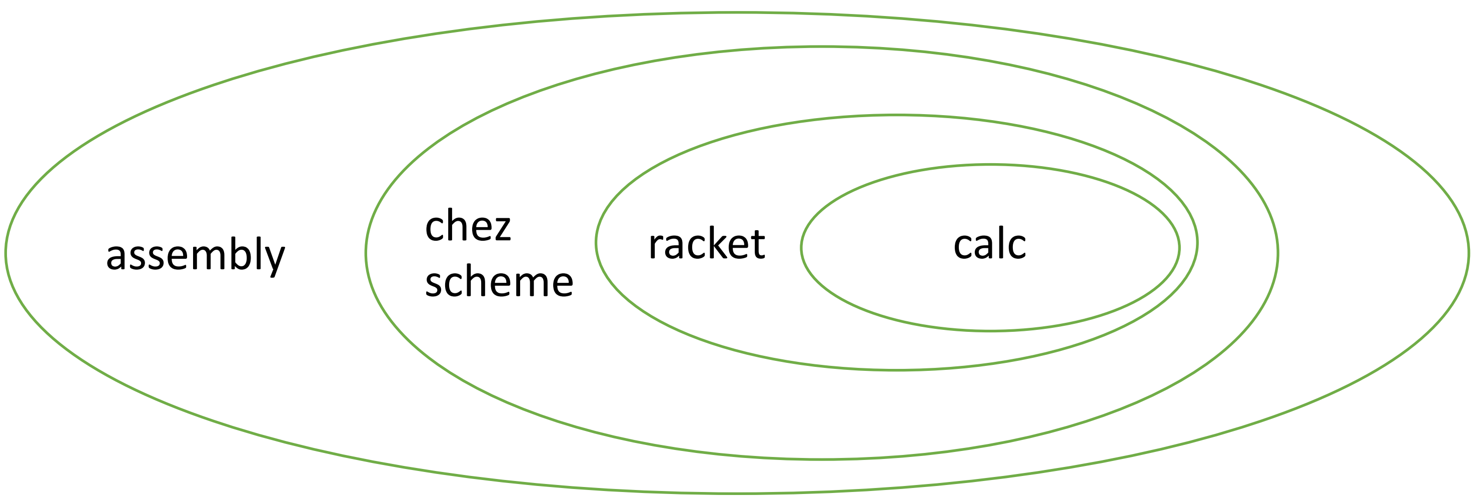
- A: A function of type
  `(Number Number -> Number)`

- Takes two arguments of Plait type `Number` and produces their arithmetic sum

  - There are alternative semantics of "+": perhaps it permits concatenating strings, etc…

Host semantics

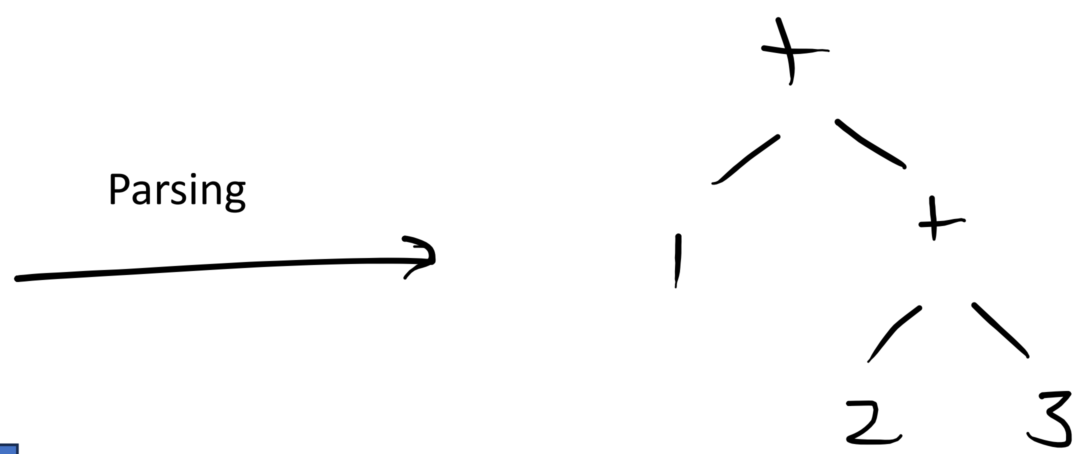
- In turn, Plait's semantics of "+" are inherited from Racket…
  - … which is inherited from Chez scheme…
  - ... which is inherited from the assembly language instruction standard for running addition…



Parsing & grammars

- Goal of parsing: given a **surface syntax** description of a program, generate an **abstract syntax tree**

`(+ 1 (+ 2 3))`



How do we describe surface syntax? We call a description of surface syntax a **grammar**

Describing grammars in English

- An `calc` expression is either:
  - A number
  - A string of the form "(" followed by "+" followed by an expression followed by an expression followed by ")"

- This gets quite unwieldy to write down as our language gets increasingly complex
- The designers of ALGOL60 agreed, so designed a system for describing surface syntax grammars called **Backus-Naur Form**

Describing Surface Syntax with Backus-Naur Form (BNF)

- BNF is a lightweight notation for describing surface syntax

```
<plus> ::= (+ <e> <e>)
<e>    ::= num | <plus>
<start> ::= <e>
```

- This grammar captures our English description of `calc` expressions
  - The symbol <e> is called a **non-terminal**
  - The symbol `num` is called a terminal: it is defined to be any syntactic Plait number (`0.1, 10, 1/2`, …
  - One symbol is designated as the start symbol

BNF and Plait Datatypes

- There is a one-to-one map between a BNF and our abstract syntax tree datatype in Plait
  - The BNF includes details about the text (i.e., contains characters)

```
<plus> ::= (+ <e> <e>)
<e>    ::= num | <plus>
<start> ::= <e>
```

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

Parsing with BNF

```
<plus>  ::= (+ <e> <e>)
<e>     ::= num | <plus>
<start> ::= <e>
```

```
<e>
```

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

Parsing with BNF

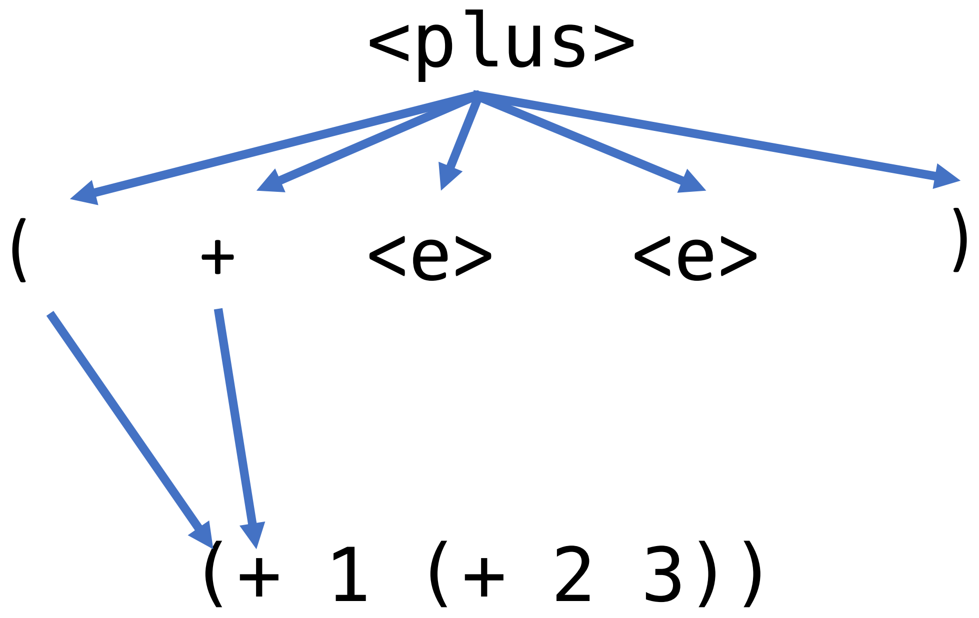
```
<plus> ::= (+ <e> <e>)
<e>    ::= num | <plus>
<start> ::= <e>
```

```
num? no
```

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

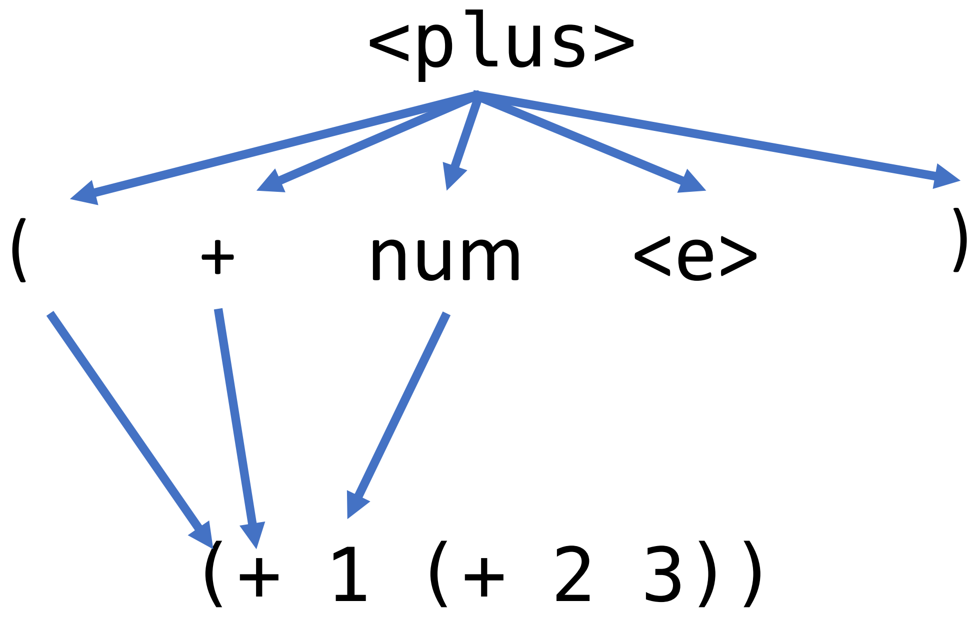
Parsing with BNF

```
<plus>  ::= (+ <e> <e>)
<e>     ::= num | <plus>
<start> ::= <e>
```



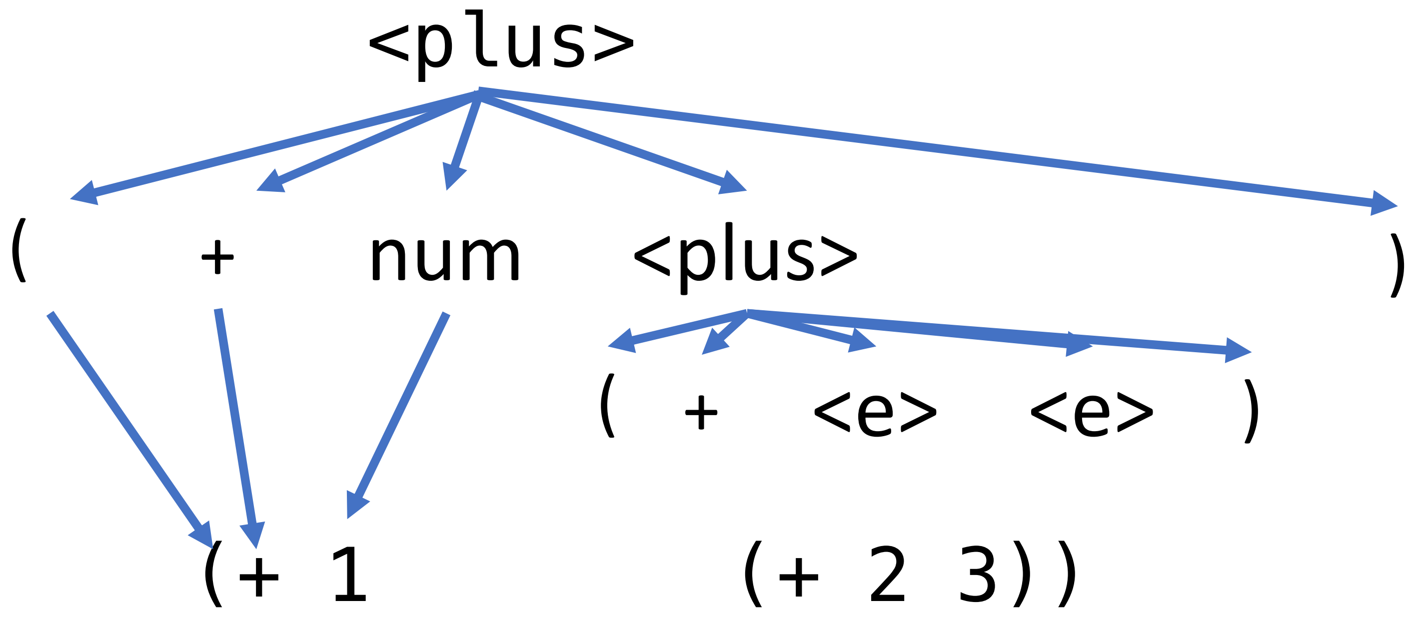
Parsing with BNF

```
<plus>  ::= (+ <e> <e>)
<e>     ::= num | <plus>
<start> ::= <e>
```



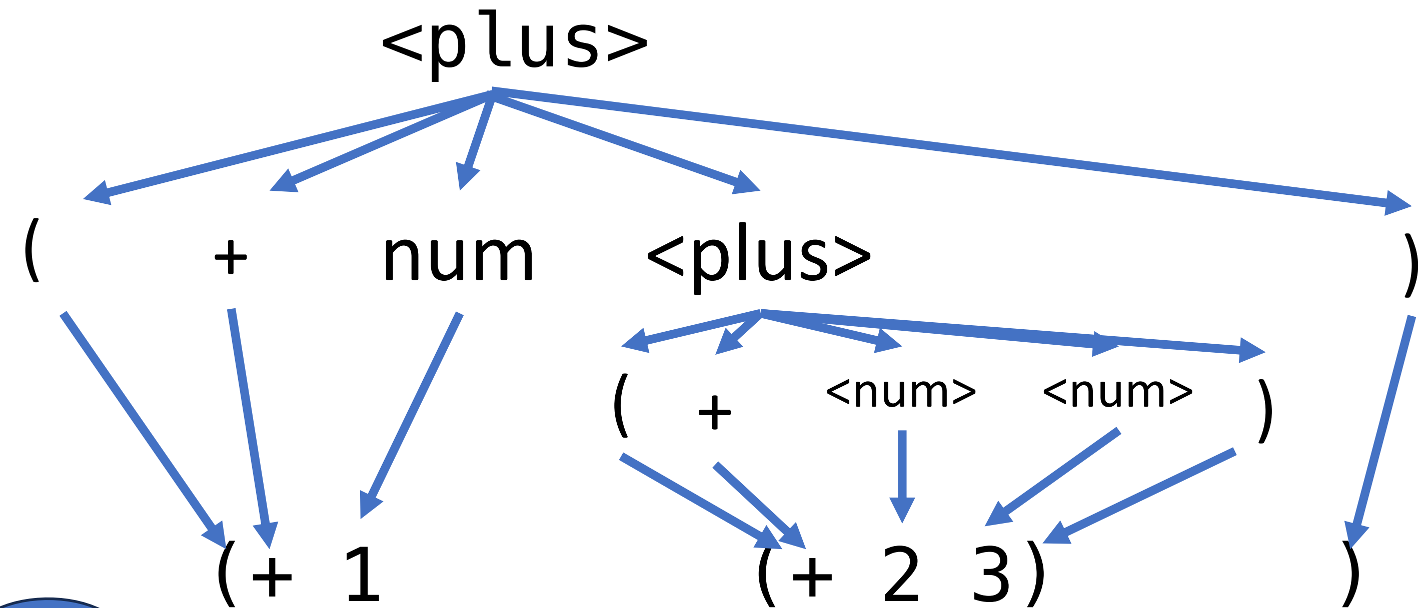
Parsing with BNF

```
<plus> ::= (+ <e> <e>)
<e>    ::= num | <plus>
<start> ::= <e>
```



Parsing with BNF

```
<plus>  ::= (+ <e> <e>)
<e>     ::= num | <plus>
<start> ::= <e>
```



This diagram is called a ***parse tree***

Parse errors

```
<plus> ::= (+ <e> <e>)
<e>    ::= num | <plus>
<start> ::= <e>
```

- A **parse error** occurs when there are no rules to apply

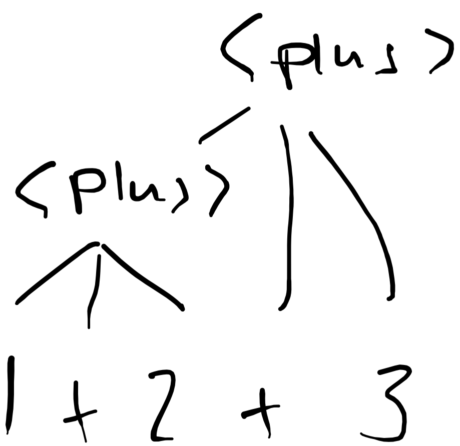
```
+ 1 2)
```

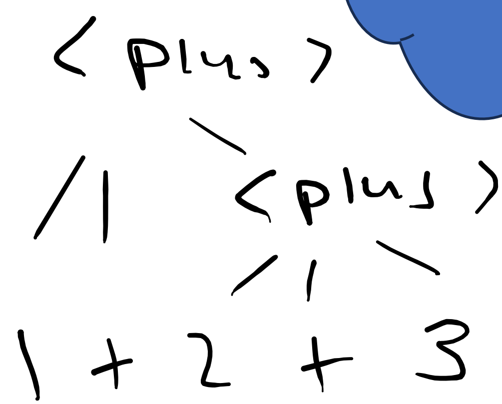
Parsing ambiguity

- Suppose we have the following grammar for infix addition:

```
<plus> ::= <e> + <e>
<e>    ::= num | <e> + <e>
```

- Then, we have two valid parse trees:





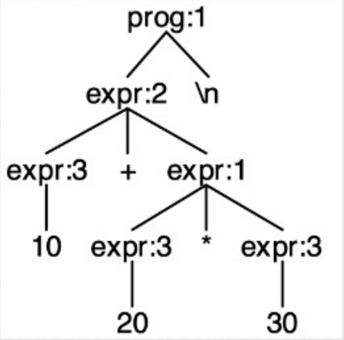
If there exists a sentence with more than 1 parse tree, we call a grammar ***ambiguous***

Parsing with BNF

- There are many tools that, given a BNF grammar, automatically generate a parser for you
- Example: antlr https://www.antlr.org/

```
grammar Expr;
prog:   (expr NEWLINE)* ;
expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   '(' expr ')'
    ;
NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
```

```
$ antlr4-parse Expr.g4 prog -gui
10+20*30
^D
$ antlr4 Expr.g4  # gen code
$ ls ExprParser.java
ExprParser.java
```



Parsing with s-expressions in Plait

- We don't need to worry about going from strings to plait datatypes: we can use Plait's parser

```
`(+ 1 2)
- S-Exp
`(+ 1 2)
```

Shorthand for:

```
(list->s-exp
  (list (symbol->s-exp '+)
        (number->s-exp 1)
        (number->s-exp 2)))
```

From s-expressions to `calc`

```
(define-type Expr
  [num (n : Number)]
  [plus (l : Expr) (r : Expr)])

(parse : (S-Exp -> Expr))
(define (parse e)
  (cond
    [(s-exp-number? e) (num (s-exp->number e))]
    [else (error 'parse "unrecognized symbol")]))
```

From s-expressions to `calc`

```
(define-type Expr
  [num (n : Number)]
  [plus (l : Expr) (r : Expr)])

(parse : (S-Exp -> Expr))
(define (parse e)
  (cond
    [(s-exp-number? e) (num (s-exp->number e))]
    [(s-exp-list? e) ???]
    [else (error 'parse "unrecognized symbol")]))
```

An aside on `let` and `local`

- Local defines a *new scope*: any definitions occurring inside local are only in scope from inside its body

List of definitions

Body

```
> (local [(define x 10)] x)
- Number
10

> (local [(define x 10) (define y 30)] (+ x y))
- Number
40

> (+ x y)
. x: free variable while typechecking in: x
```

An aside on `let` and `local`

- `let` declares local variables (short-hand for local):

List of variable declarations

```
> (let [(x 10) (y 20)] (+ x y))
- Number
30
```

Body

`letrec`

- `letrec` declares local variables that can refer to each other:

List of variable declarations

```
> (letrec [(x 10) (y (+ x 20))] (+ x y))
- Number
40
```

Body

From s-expressions to `calc`

```
(define-type Expr
  [num (n : Number)]
  [plus (l : Expr) (r : Expr)])

(parse : (S-Exp -> Expr))
(define (parse e)
  (cond
    [(s-exp-number? e) (num (s-exp->number e))]
    [(s-exp-list? e)
     (let ([l (s-exp->list e)])
       (error 'parse ""))]
    [else (error 'parse "unrecognized symbol")]))
```

From s-expressions to `calc`

It is OK in this class if your parser fails with a contract error (though, if you like, you can add specific errors for this; we won't test this

```
(define-type Expr
  [num (n : Number)]
  [plus (l : Expr) (r : Expr)])

(parse : (S-Exp -> Expr))
(parse : (S-Exp -> Expr))
(define (parse e)
  (cond
    [(s-exp-number? e) (num (s-exp->number e))]
    [(s-exp-list? e)
     (let ([l (s-exp->list e)])
       (cond
         [(empty? l) (error 'parse "empty list")]
         [(symbol=? (s-exp->symbol (first l)) '+) (plus (parse
(second l)) (parse (third l)))]
         [else (error 'parse "unrecognized symbol")]))]
    [else (error 'parse "unrecognized s-exp")]))
```

Conditionals

The language `cond`

https://gist.github.com/SHoltzen/8ec4d0ec1619a623ff8a4779072eb660

A proposed `cond AST`

```
(define-type Expr
  [num (n : Number)]
  [plus (l : Expr) (r : Expr)]
  [cnd (guard : Expr) (thn : Expr) (els : Expr)])
```

Semantics of `cond`

- Currently our calc language has a single value type: **numbers**
- One option for semantics of `cnd`: if the guard is zero, evaluate the `then` branch; else evaluate the `else` branch

```
(cnd 10 20 30) → 20
```

Many languages have somewhat interesting semantics for `if`…

main.c   Save   Run

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    // Write C code here
    if(10) {
        printf("1");
    }
    if(0) {
        printf("2");
    }
    if(-1) {
        printf("3");
    }

    return 0;
}
```

Output

```
/tmp/T9huXgalyf.o
13
```

Many languages have somewhat interesting semantics for `if`…

main.js

Run

```javascript
// Online Javascript Editor for free
// Write, Edit and Run your Javascript code using JS Online Compiler

if(0) {
    console.log("0 is true");
}
if(1) {
    console.log("1 is true");
}
if("wat") {
    console.log("hmm");
}
if("") {
    console.log("!?");
}
```

Output

```
node /tmp/M76UL60pni.js
1 is true
hmm
```

An evaluator

```
(define-type Expr
  [num (n : Number)]
  [plus (l : Expr) (r : Expr)]
  [cnd (guard : Expr) (thn : Expr) (els : Expr)])

(define (calc e)
  (type-case Expr e
    [(num v) v]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd guard thn els)
     (if (equal? 0 (calc guard))
         (calc thn)
         (calc els))]))
```

How are we using host semantics here? What are the consequences of that?

Cond with Booleans: Syntax

```
(define-type Exp
  [num (n : Number)]
  [bool (b : Boolean)]
  [plus (left : Exp) (right : Exp)]
  [cnd (test : Exp) (thn : Exp) (els : Exp)])
```

Cond with Booleans: Semantics

- If the guard evaluates to `#t`, evaluate thn; if guard evaluates to `#f`, evaluate els; otherwise, error

(if true 10 20) → 10

Developing an evaluator

- We will walk through developing this in class:

https://gist.github.com/SHoltzen/8ec4d0ec1619a623ff8a4779072eb660