Part I

An ***interpreter*** takes a program and returns it value

Plait = the language that we use to write interpreters

Curly = the language that to be interpreted

... that keeps changing

Curly Arithmetic

```
{+ 2 1}
```

$\twoheadrightarrow$ `3`

Curly Arithmetic

```
{* 2 1}
```

$\twoheadrightarrow$ `2`

Curly Arithmetic

```
{+ 2 {* 4 3}}
```

$\twoheadrightarrow$ `14`

Curly Arithmetic

$$\begin{aligned} &\mathtt{2} \\ \twoheadrightarrow\ &\mathtt{2} \end{aligned}$$

Representing Expressions

```
2

{+ 2 1}

{+ 2 {* 4 3}}
```

- numbers
- addition expressions
  - first and second arguments are expressions
- multiplication expressions
  - first and second arguments are expressions

Representing Expressions

```
2

{+ 2 1}

{+ 2 {* 4 3}}
```

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp]
         [r : Exp])
  (multE [l : Exp]
         [r : Exp]))
```

Part 2

Curly Interpeter

```
(define (interp [a : Exp]) : Number
  (type-case Exp a
    [(numE n) n]
    [(plusE l r) (+ (interp l) (interp r))]
    [(multE l r) (* (interp l) (interp r))]))

(test (interp (numE 2))
      2)
(test (interp (plusE (numE 2) (numE 1)))
      3)
(test (interp (multE (numE 2) (numE 1)))
      2)
(test (interp (plusE (multE (numE 2) (numE 3))
                     (plusE (numE 5) (numE 8))))
      19)
```

Part 3

Concrete vs. Abstract Syntax

```
{+ 2 1}
```

```
(plusE (numE 2) (numE 1))
```

Concrete Syntax as an S-Expression

```
          `{+ 2 1}

(test (parse `{+ 2 1})
      (plusE (numE 2) (numE 1)))
```

Concrete Syntax as an S-Expression

```
; An EXP is either
;  - `NUMBER
;  - `{+ EXP EXP}
;  - `{* EXP EXP}
```

Concrete Syntax as an S-Expression

```
; An EXP is either
;   - `NUMBER
;   - `{+ EXP EXP}
;   - `{* EXP EXP}
```

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp] [r : Exp])
  (multE [l : Exp] [r : Exp]))
```

Concrete Syntax as an S-Expression

```
; An EXP is either
;   - `NUMBER
;   - `{+ EXP EXP}
;   - `{* EXP EXP}
```

$\Downarrow$

parse

$\Downarrow$

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp] [r : Exp])
  (multE [l : Exp] [r : Exp]))
```

Matching an S-Expression

```
; An EXP is either ...
;  - `{* EXP EXP}
```

```
(define (parse [s : S-Exp]) : Exp
  ....




  ....)
```

Matching an S-Expression

```
; An EXP is either ...
;  - `{* EXP EXP}
```

```
(define (parse [s : S-Exp]) : Exp
  ....
  (and (s-exp-list? s)
       (= 3 (length (s-exp->list s)))
       (s-exp-symbol? (first (s-exp->list s)))
       (eq? '* (s-exp->symbol (first (s-exp->list s)))))

  ....)
```

Matching an S-Expression

```
; An EXP is either ...
;  - `{* EXP EXP}
```

```
(define (parse [s : S-Exp]) : Exp
  ....


    (s-exp-match? `{* ANY ANY} s)


  ....)
```

Matching an S-Expression

```
; An EXP is either ...
;  - `{* EXP EXP}
```

```
(define (parse [s : S-Exp]) : Exp
  ....
  (cond
    ....
    [(s-exp-match? `{* ANY ANY} s)
     .... (parse (second (s-exp->list s)))
     .... (parse (third (s-exp->list s))) ....]
    ....)
  ....)
```