Intro to Interpreters

CS 350

Dr. Joseph Eremondi

Last updated: June 24, 2024

Test

TITLE

Part 1

TITLE

- An interpreter takes a program and returns it value

TITLE

- An interpreter takes a program and returns it value
- Plait = the language that we use to write interpreters

TITLE

- 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

TITLE

- 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

Curly Arithmetic

```racket
{+ 2 1}
```

#+begin_src racket 3 #+end_src

Curly Arithmetic

Curly Arithmetic

```
{* 2 1}
```

Curly Arithmetic

```
{* 2 1}
```

```
2
```

Curly Arithmetic

Curly Arithmetic

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

Curly Arithmetic

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

```
 14
```

Curly Arithmetic

Curly Arithmetic

```
2
```

Curly Arithmetic

```
2
```

```
  2
```

Representing Expressions

Representing Expressions

```
2
{+ 2 1}
{+ 2 {* 4 3}}
```

- numbers

Representing Expressions

```
2
{+ 2 1}
{+ 2 {* 4 3}}
```

- numbers
- addition expressions

Representing Expressions

```
2
{+ 2 1}
{+ 2 {* 4 3}}
```

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

Representing Expressions

```
2
{+ 2 1}
{+ 2 {* 4 3}}
```

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

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

Representing Expressions

```
2
{+ 2 1}
{+ 2 {* 4 3}}
```

Representing Expressions

```
2
{+ 2 1}
{+ 2 {* 4 3}}
```

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

TITLE

Part 2

Curly Interpeter

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)
```

TITLE

Part 3

Concrete vs. Abstract Syntax

Concrete vs. Abstract Syntax

```
{+ 2 1}
```

Concrete vs. Abstract Syntax

```
{+ 2 1}
```

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

Concrete Syntax as an S-Expression

Concrete Syntax as an S-Expression

```
`{+ 2 1}
```

Concrete Syntax as an S-Expression

```
`{+ 2 1}
```

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

Concrete Syntax as an S-Expression

Concrete Syntax as an S-Expression

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

Concrete Syntax as an S-Expression

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

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}
```

```
parse
```

Concrete Syntax as an S-Expression

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

```
parse
```

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

Matching an S-Expression

Matching an S-Expression

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

Matching an S-Expression

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

```
(define (parse [s : S-Exp]) : Exp
....
#+begin_src racket
```

....) #+end_src

Matching an S-Expression

Matching an S-Expression

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

Matching an S-Expression

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

```
(define (parse [s : S-Exp]) : Exp
....
#+begin_src racket
(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)))))
```

....) #+end_src

Matching an S-Expression

Matching an S-Expression

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

Matching an S-Expression

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

```
(define (parse [s : S-Exp]) : Exp
....
#+begin_src racket
(s-exp-match? `{* ANY ANY} s)
```

….) #+end_src

Matching an S-Expression

Matching an S-Expression

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

Matching an S-Expression

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

```
(define (parse [s : S-Exp]) : Exp
....
#+begin_src racket
(cond
....
[#+begin_src racket
(s-exp-match? `{* ANY ANY} s)
```

.... (parse (second (s-exp->list s))) .... (parse (third (s-exp->list s))) ....] ....) #+end_src ....) #+end_src