Outline

Interpreting Language B using Language A

Macros

Quoting & Self Interpretation
Building an Interpreter for B
Assumptions, Semantics, and Evaluation
Building an Interpreter for B
Assumptions, Semantics, and Evaluation

- We are skipping the parsing phase ← Do Not Implement
Building an Interpreter for \textbf{B}
Assumptions, Semantics, and Evaluation

- We are skipping the parsing phase \hspace{1em} \textcolor{red}{\leftarrow \textit{Do Not Implement}}
- Interpreter is written in \textbf{Racket}
  - Racket in this case is the \textit{metalanguage} \textbf{A}
Building an Interpreter for B
Assumptions, Semantics, and Evaluation

• We are skipping the parsing phase ← Do Not Implement
• Interpreter is written in Racket
  • Racket in this case is the metalanguage A
• Language B syntax will be represented with an AST
  • AST nodes made up of B’s constructors will be structs to Racket
  • Allows us to skip the parsing stage (it’s already parsed this way!)
Building an Interpreter for B
Assumptions, Semantics, and Evaluation

• We are skipping the parsing phase ← Do Not Implement
• Interpreter is written in Racket
  • Racket in this case is the metalanguage A
• Language B syntax will be represented with an AST
  • AST nodes made up of B’s constructors will be structs to Racket
  • Allows us to skip the parsing stage (it’s already parsed this way!)
• You assume AST input has valid syntax
Building an Interpreter for B
Assumptions, Semantics, and Evaluation

- We are skipping the parsing phase ← Do Not Implement
- Interpreter is written in Racket
  - Racket in this case is the metalanguage A
- Language B syntax will be represented with an AST
  - AST nodes made up of B’s constructors will be structs to Racket
  - Allows us to skip the parsing stage (it’s already parsed this way!)
- You assume AST input has valid syntax
- You cannot assume an AST has correct semantics
Building an Interpreter for B
Correct Syntax Examples

Using these Racket structs (i.e. using syntax and semantics of A):

```
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)
```

We can interpret programs written in B:

```
(int 34)
(add (int 34) (int 30))
(ifnz (add (int 5) (int 7)) (int 12) (int 1))
```
Building an Interpreter for B
Incorrect Syntax Examples

Using these Racket structs (i.e. using syntax and semantics of A):

(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)

You can assume you won’t see programs in B like this:

(int "dan then dog")
(int (ifnz (int 0) (int 5) (int 7)))
(add (int 8) #t)
(add 5 4)
Building an Interpreter for B
Language A vs. Language B

In Racket, our language A, structs can take any Racket value:

(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)

But in B, we restrict int to take only an integer value, add to take two B expressions, and so on:

(int "dan then dog")
(int (ifnz (int 0) (int 5) (int 7)))
(add (int 8) #t)
(add 5 4)
Building an Interpreter for B
Language A vs. Language B

In Racket, our language A, structs can take *any* Racket value:

```plaintext
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)
```

But in B, we restrict `int` to take only an integer value, add to take two B expressions, and so on:

```plaintext
(int "dan then dog")
(int (ifnz (int 0) (int 5) (int 7)))
(add (int 8) #t)
(add 5 4)
```

So the above is valid syntax in Racket, but not valid syntax for B
Building an Interpreter for B
Language A vs. Language B

In Racket, our language A, structs can take *any* Racket value:

```
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)
```

But in B, we restrict int to take only an integer value, add to take two B expressions, and so on:

```
(int "dan then dog")
(int (ifnz (int 0) (int 5) (int 7)))
(add (int 8) #t)
(add 5 4)
```

So the above is valid syntax in Racket, but not valid syntax for B

*Illegal input ASTs may crash the interpreter; this is OK*
Building an Interpreter for B

Evaluating the AST

- **eval-exp** should return a value of language B
- Values in language B evaluate to themselves
- Otherwise, we have an unsimplified expression in B
Building an Interpreter for $\mathbf{B}$

Evaluating the AST

- **eval-exp** should return a value of language $\mathbf{B}$
- Values in language $\mathbf{B}$ evaluate to themselves
- Otherwise, we have an unsimplified expression in $\mathbf{B}$

(int 7); evaluates to (int 7)
(add (int 3) (int 4)); evaluates to (int 7)
What if the program is a valid AST, but evaluation of it tries to use the \textit{wrong} kind of value?
Building an Interpreter for B
Checking for Correct Semantics

What if the program is a valid AST, but evaluation of it tries to use the wrong kind of value?

(add (int 3) (bool #f)) ; evaluates to ?
Building an Interpreter for B
Checking for Correct Semantics

What if the program is a valid AST, but evaluation of it tries to use the *wrong* kind of value?

```
(add (int 3) (bool #f)) ; evaluates to ?
```

You should detect this and give an error message that is not in terms of the interpreter implementation
What if the program is a valid AST, but evaluation of it tries to use the wrong kind of value?

```
(add (int 3) (bool #f)) ; evaluates to ?
```

You should detect this and give an error message that is not in terms of the interpreter implementation

We need to check that the type of a recursive result is what we expect

- No need to check if any type is acceptable
1. Extend language syntax
2. Written in terms of **existing syntax**
3. Expanded before language is actually interpreted or compiled
   - The macro itself is *never* evaluated beyond its replacement with different syntax
Macros for Language B

- Interpreting B using Racket as the metalanguage A

\[
\text{(define (++ exp) (add (int 1) exp))}
\]

This extends language B to have the syntax \((++ \text{exp})\) where \text{exp} is an expression in B.
Macros for Language $\mathbf{B}$

- Interpreting $\mathbf{B}$ using Racket as the metalanguage $\mathbf{A}$
- Language $\mathbf{B}$ is made up of Racket structs

(define (++ exp) (add (int 1) exp))

This extends language $\mathbf{B}$ to have the syntax $(++ \text{exp})$ where $\text{exp}$ is an expression in $\mathbf{B}$.
Macros for Language B

- Interpreting B using Racket as the metalanguage A
- Language B is made up of Racket structs
- Why not write a Racket function that returns ASTs in the syntax of language B?
Macros for Language B

- Interpreting B using Racket as the metalanguage A
- Language B is made up of Racket structs
- Why not write a Racket function that returns ASTs in the syntax of language B?

Define macros for B using Racket functions

```
(define (++ exp) (add (int 1) exp))
```

This extends language B to have the syntax (++ exp) where exp is an expression in B
Macros for Language \textbf{B}

Define macros for \textbf{B} using Racket functions

\begin{verbatim}
(define (++ exp) (add (int 1) exp))
\end{verbatim}

This \textit{extends} language \textbf{B} to have the syntax \((++ \ exp)\) where \(\text{exp}\) is an expression in \textbf{B}.

What happens when we use \((++ \ exp)\) when writing code in language \textbf{B}?
Macros for Language B

Define macros for B using Racket functions

\[
(\text{define } (++ \text{ exp}) (\text{add } (\text{int } 1) \text{ exp}))
\]

This *extends* language B to have the syntax \((++ \text{ exp})\) where \text{exp} is an expression in B.

What happens when we use \((++ \text{ exp})\) when writing code in language B?

- Replace with *existing syntax* in language B: \((\text{add } (\text{int } 1) \text{ exp})\)
Define macros for B using Racket functions

```
(define (++ exp) (add (int 1) exp))
```

This *extends* language B to have the syntax `(++ exp)` where `exp` is an expression in B.

What happens when we use `(++ exp)` when writing code in language B?

- Replace with **existing syntax** in language B: `(add (int 1) exp)`
  - This replacement is done by *evaluating* the Racket function in Racket
Macros for Language $\mathbf{B}$

Define macros for $\mathbf{B}$ using Racket functions

\[
\text{(define (++ exp) (add (int 1) exp))}
\]

This *extends* language $\mathbf{B}$ to have the syntax $(++ \ exp)$ where $\exp$ is an expression in $\mathbf{B}$

What happens when we use $(++ \ exp)$ when writing code in language $\mathbf{B}$?

- Replace with *existing syntax* in language $\mathbf{B}$: $(\text{add (int 1) exp})$
  - This replacement is done by *evaluating* the Racket function in Racket
- Evaluate the resulting language $\mathbf{B}$ code
Macros for Language B

Define macros for B using Racket functions

```
(define (++ exp) (add (int 1) exp))
```

This extends language B to have the syntax (++ exp) where exp is an expression in B

What happens when we use (++ exp) when writing code in language B?

- Replace with **existing syntax** in language B: (add (int 1) exp)
  - This replacement is done by *evaluating* the Racket function in Racket
- Evaluate the resulting language B code

Is this any different from macros as we know them?
Macros for Language B

Define macros for B using Racket functions

```
(define (++ exp) (add (int 1) exp))
```

This extends language B to have the syntax (++ exp) where exp is an expression in B

What happens when we use (++ exp) when writing code in language B?
- Replace with existing syntax in language B: (add (int 1) exp)
  - This replacement is done by evaluating the Racket function in Racket
- Evaluate the resulting language B code

Is this any different from macros as we know them?
- No! Clients have no idea how the replacement is being done
• Syntactically, Racket statements can be thought of as lists of tokens
• (+ 3 4) is a “plus sign”, a “3”, and a “4”
• quote-ing a parenthesized expression produces a list of tokens

Examples:

(+ 3 4) ; 7
(quote (+ 3 4)) ; ’(+ 3 4)
(quote (+ 3 #t)) ; ’(+ 3 #t)
(+ 3 #t) ; Error

Syntactic sugar for quoting and evaluation exists (use ‘ instead of quote) but we won’t get into it
Quasiquote

Allows evaluation of particular tokens into a quote

(quote (+ 3 (+ 2 2))) ; (list ’+ ’3 ’(+ 2 2))
(quasiquote (+ 3 (unquote(+ 2 2)))) ; (list ’+ ’3 ’4)
Quasiquote

Allows evaluation of particular tokens into a quote

(quote (+ 3 (+ 2 2))) ; (list '+ '3 '(+ 2 2))
(quasiquote (+ 3 (unquote(+ 2 2)))) ; (list '+ '3 '4)

- Convenient for generating dynamic token lists
- Use unquote to escape a quasiquote back to evaluated Racket code
- A quasiquote and quote are equivalent unless we use an unquote operation
Quasiquote

Allows evaluation of particular tokens into a quote

(quote (+ 3 (+ 2 2))) ; (list '+ '3 '(+ 2 2))
(quasiquote (+ 3 (unquote(+ 2 2)))) ; (list '+ '3 '4)

- Convenient for generating dynamic token lists
- Use unquote to escape a quasiquote back to evaluated Racket code
- A quasiquote and quote are equivalent unless we use an unquote operation

(quasiquote
  (string-append
   "I love CSE"
   (number->string
     (unquote (+ 3 338)))))

; '(string-append "I love CSE" (number->string 341))
Self Interpretation

• Many languages provide an eval function or something similar
• Performs interpretation or compilation at **runtime**
  • But needs the full language implementation at runtime
• It’s useful, but there’s usually a better way
• Makes analysis, debugging difficult
Racket’s `eval` operates on lists of tokens
- Like those generated from `quote` and `quasiquote`
- Treat the input data as a program and evaluate it

```scheme
(define quoted (quote (+ 3 4)))
(eval quoted)
(define bad-quoted (quote (+ 3 #t)))
(eval bad-quoted)
(define qquoted (quasiquote (+ 3 (unquote (+ 2 2)))))
(eval qquoted)
(define big-qquoted
  (quasiquote
    (string-append
      "I love CSE"
      (number->string
        (unquote (+ 3 338)))))
(eval big-qquoted)
```
Variable Number of Arguments

- Some functions (like +) can take a variable number of arguments
- There is syntax that lets you define your own

```scheme
(define fn-any
  (lambda xs ; any number of args
    (print xs)))

(define fn-1-or-more
  (lambda (a . xs) ; at least 1 arg
    (begin (print a) (print xs)))

(define fn-2-or-more
  (lambda (a b . xs) ; at least 2 args
    (begin (print a) (print a) (print xs))))
```
apply applies a list of values as the arguments to a function in order by position

```
(define fn-any
  (lambda xs ; any number of args
    (print xs)))
(apply fn-any (list 1 2 3 4))

(apply + (list 1 2 3 4)) ; 10
(apply max (list 1 2 3 4)) ; 4
```