CMSC 430, Feb 25th 2020

Grift
Update
Update

• I have credentials!
Update

• I have credentials!
  ○ I got UMD credentials near the end of last week, and am now jumping through all the hoops to get your grades on ELMS/Canvas
Update

• I have credentials!
  ○ I got UMD credentials near the end of last week, and am now jumping through all the hoops to get your grades on ELMS/Canvas
  ○ Hoping to get grading done by the end of the week.
Update

• I have credentials!
  ○ I got UMD credentials near the end of last week, and am now jumping through all the hoops to get your grades on ELMS/Canvas
  ○ Hoping to get grading done by the end of the week.
  ○ I’m scraping the plan of having the TA disambiguate and am going to try and do it through ELMS. You should already see a quiz on ELMS?
Word to the wise
Word to the wise

• I’ve been seein some of the assignments that have been submitted
Word to the wise

• I’ve been seein some of the assignments that have been submitted
  ○ I am consistently seeing a very serious mistake!
Word to the wise

• I’ve been seein some of the assignments that have been submitted
  ○ I am consistently seeing a very serious mistake!
  ○ Unless you defined `interp` using macros, you must quote your input expression!
Word to the wise

• I’ve been seein some of the assignments that have been submitted
  ○ I am consistently seeing a very serious mistake!
  ○ Unless you defined `interp` using macros, *you must quote your input expression!*
  ○ Why is the following wrong?
Word to the wise

• I’ve been seein some of the assignments that have been submitted
  ◦ I am consistently seeing a very serious mistake!
  ◦ Unless you defined \texttt{interp} using macros, \textit{you must quote your input expression}!
  ◦ Why is the following wrong?

\begin{verbatim}
(check-equal? (interp (add1 1)) 2)
\end{verbatim}
Word to the wise
Word to the wise

• This is partly my fault (and is why I stopped using macros in class)
Word to the wise

• This is partly my fault (and is why I stopped using macros in class)
  ◦ Some of you wrote tests (yay!)
Word to the wise

- This is partly my fault (and is why I stopped using macros in class)
  - Some of you wrote tests (yay!)
  - But those tests are just testing racket, not your interpreter.
Word to the wise

• This is partly my fault (and is why I stopped using macros in class)
  ○ Some of you wrote tests (yay!)
  ○ But those tests are just testing racket, not your interpreter.
  ○ This is not a rare mistake. You should _all_ double-check your code.
Appreciating what we have:
Appreciating what we have:

• To recap, we’ve got:
Appreciating what we have:

• To recap, we’ve got:
  ○ *unary* arithmetic primitives
Appreciating what we have:

• To recap, we’ve got:
  ○ *unary* arithmetic primitives
  ○ Conditionals, for branching
Appreciating what we have:

• To recap, we’ve got:
  ○ *unary* arithmetic primitives
  ○ Conditionals, for branching
  ○ Errors that halt our programs
Appreciating what we have:

• To recap, we’ve got:
  ○ *unary* arithmetic primitives
  ○ Conditionals, for branching
  ○ Errors that halt our programs
  ○ let-bound variables
Grift
Grift

- What would be useful to add?
Fraud’s AST

\[ e = i \mid b \mid \text{if } e\ e\ e\ e \mid \text{let } ((\text{id } e))\ e \mid \text{id} \mid p\ e \]
Fraud’s AST

\[ e = i \mid b \mid \text{if } e \ e \ e \mid \text{let } ((\text{id } e)) \ e \mid \text{id} \mid p \ e \]

\[ p = \text{add1} \mid \text{sub1} \mid \text{zero}? \]
Fraud’s AST

- $e = i \mid b \mid if \ e \ e \ e \mid let ((id \ e)) \ e \mid id \mid p \ e$
  - $p = add1 \mid sub1 \mid zero?$
- $id = variable$
Grift’s AST

• We go
• from:
Grift’s AST

- We go
- from:
  - $e = \ldots | p\ e$
Grift’s AST

• We go

• to:
Grift’s AST

• We go
• to:
  • $e = \ldots \mid p1 \ e \mid p2 \ e \ e$
Grift’s AST

- We go
- to:
  - $e = \ldots | p1\ e | p2\ e\ e$
  - $p1 = \text{add1} | \text{sub1} | \text{zero}$
Grift’s AST

• We go
to:
  ○ $e = \ldots \mid p1 \ e \mid p2 \ e \ e$
  ○ $p1 = \text{add1} \mid \text{sub1} \mid \text{zero}$
  ○ $p2 = + \mid -$
Binary Operators!
Binary Operators!

• Interpretation is easy (as we’ll see)
Binary Operators!

- Interpretation is easy (as we’ll see)
- Compilation is not hard, but requires a non-trivial insight (as we’ll see)
Binary Operators!

• Interpretation is easy (as we’ll see)
• Compilation is not hard, but requires a non-trivial insight (as we’ll see)
• Can anyone think of why interpretation might be much easier?
Meanings
Meanings

• Grift doesn’t add much:
Meanings

• Grift doesn’t add much:

• First we factor out a rule for primitives

\[
\begin{align*}
G-env[e_0, r, a_0] & \quad \ldots \\
\hline
G-env[(p e_0 \ldots), r, G-prim[(p a_0 \ldots)] & 
\end{align*}
\]
Meanings

• Grift doesn’t add much:

• Then we use that rule

\[
\begin{align*}
\text{G-prim} & : (p \ a \ \ldots) \rightarrow a \\
\text{G-prim}[(p \ \lor \ \ldots \ \text{err} \ \_ \ \ldots)] & = \text{err} \\
\text{G-prim}[(\text{add1} \ i_0)] & = (+ \ i_0 \ 1) \\
\text{G-prim}[(\text{sub1} \ i_0)] & = (- \ i_0 \ 1) \\
\text{G-prim}[(\text{zero?} \ 0)] & = \#t \\
\text{G-prim}[(\text{zero?} \ i)] & = \#f \\
\text{G-prim}[(+ \ i_0 \ i_1)] & = (+ \ i_0 \ i_1) \\
\text{G-prim}[(\text{-} \ i_0 \ i_1)] & = (- \ i_0 \ i_1) \\
\text{G-prim}[\_\_\_] & = \text{err}
\end{align*}
\]
Interpreter
Interpreter

• Switch to the terminal...
The Compiler
The Compiler

- We can't do it naively, consider:
• We can’t do it naively, consider:

```
(define (compile-+ e0 e1 c)
  (let ((c0 (compile-e e0 c))
        (c1 (compile-e e1 c)))
    `(
      ,@c0
      ,@c1
      (add rax ???)))
```
The Compiler
The Compiler

• What are some alternatives?
The Compiler

• What are some alternatives?
• With those alternatives in mind, consider:
The Compiler

• What are some alternatives?
• With those alternatives in mind, consider:

\[
(+ (\text{add1} 2) (\text{add1} 3))
\]
The Compiler

• What are some alternatives?
• With those alternatives in mind, consider:

\[
(+ \ (\text{add1} \ 2) \ (\text{add1} \ 3))
\]

\[
(+ \ (\text{add1} \ 2) \ 3)
\]
• What are some alternatives?
• With those alternatives in mind, consider:

\[
\begin{align*}
(+ \ (\text{add1} \ 2) \ (\text{add1} \ 3)) \\
(+ \ (\text{add1} \ 2) \ 3) \\
(+ \ (\text{add1} \ 2) \ x)
\end{align*}
\]
The Compiler
The Compiler

• Before we dive in, let’s review compiling `let` and add comments
The Compiler

• Before we dive in, let’s review compiling `let` and add comments
  ○ Reminder to José: in assembly they’re called `remarks`