Interpreting IfArith

CIS352 — Spring 2021
Kris Micinski
Bug in P2 public testcases (TC)

• Very sorry about that—bonus portion I didn’t assign was buggy (caused tests to be buggy)

• No changes needed to starter code—check out p2-fix from autograde

• P2 is blackholed, you can’t submit—but you can copy-paste your solution to p2-fix

• Deadline extended to Oct 15 (Saturday) since it was my fault
Today, we’re going to start building our own languages

We’re going to do this by writing interpreters
To build a programming language, we need two things:

A **syntax** for the language (and the ability to **parse** it)

A **semantics** for the language. Typically either an **interpreter** or a **compiler**
For this class, all of our programs are going to be written as Racket datums.

We specify syntax via a predicate that uses pattern matching.

This means we can just write programs in our language just by building data in Racket.
Here is the first language we will define:

(define (expr? e)
  (match e
      [(? integer? n) #t]
      [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
      [`(div ,(? expr? e0) ,(? expr? e1)) #t]
      [`(not ,(? expr? e-guard)) #t]
      [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
      [_ #f]]
)
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]])

"Any integer is a program in our language."
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]]
)

"If e0 is an expression in our language, and e1 is an expression in our language, `(plus ,e0 ,e1) is, too."
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]])

Here are some example expressions:
'(plus 1 (div 2 3))
'(if 0 (plus 1 2) (div 2 2))
'(if 0 (plus 1 (div 2 3)) (if 1 (plus 2 3) 0))
IMPORTANT NOTE

We are defining a new language by using Racket. But our language is not Racket. In Racket, booleans are #t and #f. In our language, we will use 0 to represent false and non-0 to represent true (as in C).
Again, because this is confusing

When writing interpreters, always be careful to mentally separate the **language you are defining** and the language you are using to build the interpreter (Racket).

This can become confusing as the languages we build will “look like” Racket. Try to be mindful.
Key idea: write an \texttt{interp} function that takes in expressions as an argument, and returns \texttt{Racket} values
Key idea: write an `interp` function that takes in expressions as an argument, and returns Racket values.

The “result” of programs will be a Racket integer:

```scheme
(define value? integer?)
```
Key idea: write an `interp` function that takes in expressions as an argument, and returns Racket values.

The “result” of programs will be a Racket integer:

```
(define value? integer?)

(define/contract (evaluate e)
  (-> expr? value?)
  ‘todo)
```
What should the following return...?

Remember, this is our own **new language we are defining, not necessarily Racket**

(\texttt{evaluate }'(\texttt{plus 1 2}))
\Rightarrow 3

(\texttt{evaluate }'(\texttt{if 0 (plus 1 2) (div 2 2)}))
\Rightarrow 'todo

(\texttt{evaluate }'(\texttt{if 1 (div 4 3) (plus 1 -1)}))
\Rightarrow 'todo
What should the following return...?

Remember, this is our own new language we are defining, not necessarily Racket

\[
\text{(evaluate '}(\text{plus } 1 2))
\]
\[=> 3\]
\[
\text{(evaluate '}(\text{if } 0 (\text{plus } 1 2) (\text{div } 2 2)))
\]
\[=> 1\]
\[
\text{(evaluate '}(\text{if } 1 (\text{div } 4 3) (\text{plus } 1 -1)))
\]
\[=> 4/3\]
Now, let’s build evaluate ourselves
In this lecture, we built a metacircular interpreter

**Important Definition**
A metacircular interpreter is an interpreter which uses features of a “host” language to define the semantics of a “target” language

Which features of Racket did we use to define our language...?
Important Definition
A metacircular interpreter is an interpreter which uses features of a “host” language to define the semantics of a “target” language

(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
...

Notice how we inherit the definition of + from Racket
John Reynolds introduced metacircular interpreters in 1978. One key idea: metacircular interpreters inherit properties of their host language!
Note: our interpreter is **direct-style**, it is **not** tail recursive

```scheme
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    ...
)
```

This means we are relying on Racket’s **stack** as well.
We will later see how to eliminate the need for this
In this lecture, we’ll introduce natural deduction

Natural deduction is a mathematical formalism that helps ground the ideas in metacircular interpreters
Natural deduction first used in mathematical logic, to specify **proofs** using inductive data

We will use natural deduction as a framework for specifying semantics of various languages throughout the course

<table>
<thead>
<tr>
<th>Introduction Rules</th>
<th>Elimination Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Introduction Rules" /></td>
<td><img src="image" alt="Elimination Rules" /></td>
</tr>
</tbody>
</table>
When we specify the semantics of a language using natural deduction, we give its semantics via a set of inference rules.
Rules read: if the thing on the top is true, then the thing on the bottom is also true.

This rule says: “if $c$ is an integer (mathematically: $c \in \mathbb{Q}$), then $c$ evaluates to $c$.”

\[
\text{Const : } \frac{c \in \mathbb{Q}}{c \downarrow c}
\]

Note: the notation $e \downarrow v$ is read “$e$ evaluates to $v$.”
Some rules will have more than one **antecedent** (thing on the top).

You read these: “if the first thing, and second thing, and ... are **all** true, then the thing on the bottom is true.”

$$\text{Plus : } e_0 \downarrow n_0 \ e_1 \downarrow n_1 \ n' = n_0 + n_1$$

$$(\text{plus } e_0 \ e_1) \downarrow n'$$
"If $e_0 \downarrow n_0$, and $e_1 \downarrow n_1$, and $n' = n_0 + n_1$, then I can say
(plus $e_0 e_1$) $\downarrow n'$.”
**Const**: \[ c \in \mathbb{Q} \quad \frac{c \downarrow c}{c} \]

**Plus**: \[ \frac{e_0 \downarrow n_0 \quad e_1 \downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus} \ e_0 \ e_1) \downarrow n'} \]

**Div**: \[ \frac{e_0 \downarrow n_0 \quad e_1 \downarrow n_1 \quad n' = n_0/n_1}{(\text{div} \ e_0 \ e_1) \downarrow n'} \]

The natural deduction rule for **div** is similar
Const: \( c \in \mathbb{Q} \)  \( \vdash c \)

Plus: \( e_0 \downarrow n_0 \quad e_1 \downarrow n_1 \quad n' = n_0 + n_1 \)

(plus \( e_0 \quad e_1 \) \( \downarrow n' \))

Div: \( e_0 \downarrow n_0 \quad e_1 \downarrow n_1 \quad n' = n_0/n_1 \)

(div \( e_0 \quad e_1 \) \( \downarrow n' \))

Not\(_0\) : \( e \downarrow 0 \)

(not \( e \) \( \downarrow 1 \))

Not\(_1\) : \( e \downarrow n \quad n \neq 0 \)

(not \( e \) \( \downarrow 0 \))

We have two rules for not
Natural Deduction Rules for IfArith

**Const:** \( c \in \mathbb{Q} \)  
\[
\frac{c \downarrow c}{c}
\]

**Plus:**  
\[
\frac{e_0 \downarrow n_0 \quad e_1 \downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0 \ e_1) \downarrow n'}
\]

**Div:**  
\[
\frac{e_0 \downarrow n_0 \quad e_1 \downarrow n_1 \quad n' = n_0 / n_1}{(\text{div } e_0 \ e_1) \downarrow n'}
\]

**Not\_0:**  
\[
\frac{e \downarrow 0}{(\text{not } e) \downarrow 1}
\]

**Not\_1:**  
\[
\frac{e \downarrow n \quad n \neq 0}{(\text{not } e) \downarrow 0}
\]

**If\_T:**  
\[
\frac{e_0 \downarrow 0 \quad e_1 \downarrow n'}{(\text{if } e_0 \ e_1 \ e_2) \downarrow n'}
\]

**If\_F:**  
\[
\frac{e_0 \downarrow n \quad n = 0 \quad e_2 \downarrow n'}{(\text{if } e_0 \ e_1 \ e_2) \downarrow n'}
\]
Question: Now that we have the rules, what can we do with them?

Answer: Use them to **formally prove** that some program calculates some result
Let’s say I want to prove that the following program evaluates to 4:

(if (plus 1 -1) 3 4)
What rule could go here..?

\[
\frac{???
}{(\text{if } (\text{plus } 1 \ - \ 1) \ 3 \ 4) \ \downarrow \ 4
}
If\( T: \) \[ \frac{e_0 \downarrow n \ n \neq 0 \ e_1 \downarrow n'}{(\text{if } e_0 \ e_1 \ e_2) \downarrow n'} \] If\( F: \) \[ \frac{e_0 \downarrow 0 \ e_2 \downarrow n'}{(\text{if } e_0 \ e_1 \ e_2) \downarrow n'} \] 

\[ \text{???
}\]

\[ (\text{if (plus 1 \ -1) 3 4) \downarrow 4} \]
**If** : \( e_0 \downarrow n \quad n \neq 0 \quad e_1 \downarrow n' \)  
\( \text{if } (e_0 \quad e_1 \quad e_2) \downarrow n' \)

**If** : \( e_0 \downarrow 0 \quad e_2 \downarrow n' \)  
\( \text{if } (e_0 \quad e_1 \quad e_2) \downarrow n' \)

To apply a natural-deduction rule, we must perform **unification**

There can be no variables in the resulting unification!
\textbf{If} _F: \quad \frac{e_0 \downarrow 0 \quad e_2 \downarrow n'}{(\text{if } e_0 \quad e_1 \quad e_2) \downarrow n'}

(\text{plus} \ 1 \ - \ 1) \downarrow 0 \quad 4 \downarrow 4

(\text{if} \ (\text{plus} \ 1 \ - \ 1) \ 3 \ 4) \downarrow 4

We perform unification:

e_0: (\text{plus} \ 1 \ - \ 1), \ e_1: 3
\ e_2: 4, \ n': 4
Not done yet, now we have to prove these things

\[
\frac{(\text{plus } 1 - 1) \downarrow 0 \quad 4 \downarrow 4}{(\text{if } (\text{plus } 1 - 1) \ 3 \ 4) \downarrow 4}
\]
Why can we say $4 \downarrow 4$? Because of the \textbf{Const} rule

\[
\begin{align*}
(\text{plus } 1 & \ - \ 1) \downarrow 0 \quad \frac{4 \in \mathbb{Q}}{4 \downarrow 4} \\
(\text{if } (\text{plus } 1 & \ - \ 1) \ 3 \ 4) \downarrow 4
\end{align*}
\]
We’re not done yet, because \textbf{plus} requires an antecedent:

\[
\text{Plus} : \frac{\downarrow e_0 \downarrow n_0 \quad \downarrow e_1 \downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus} \ e_0 \ e_1) \downarrow n'}
\]

\[
(\text{plus} \ 1 \ - \ 1) \downarrow 0 \quad \frac{4 \in \mathbb{Q}}{4 \downarrow 4}
\]

\[
(\text{if} \ (\text{plus} \ 1 \ - \ 1) \ 3 \ 4) \downarrow 4
\]
But we’re **still** not done, because we need to finish these three

\[
\begin{align*}
1 \downarrow 1 & - 1 \downarrow -1 & 1 + -1 = 0 \\
\text{(plus } 1 - 1) \downarrow 0 & \\
\text{(if (plus } 1 - 1) 3 4) \downarrow 4 \\
\end{align*}
\]
Things that are simply true from algebra require no antecedents, we take them as “axioms.”

\[
\begin{align*}
1 & \in \mathbb{Q} \\
\downarrow & 1 \\
-1 & \in \mathbb{Q} \\
\downarrow & -1 \\
1 + (-1) & = 0 \\
\downarrow & 0 \\
\end{align*}
\]

\[
(\text{if } (\text{plus } 1 \text{ } - 1) \downarrow 0) \downarrow 4
\]

\[
4 \in \mathbb{Q} \\
\downarrow 4
\]
This is a complete proof that the program computes 4

\[
\begin{align*}
\frac{1 \in \mathbb{Q}}{1 \downarrow 1} & \quad \frac{-1 \in \mathbb{Q}}{-1 \downarrow -1} & \quad 1 + (-1) = 0 & \quad 4 \in \mathbb{Q} \\
& & & 4 \downarrow 4 \\
\text{(plus } 1 - 1) \downarrow 0 & & & 4 \downarrow 4 \\
\text{(if (plus } 1 - 1) \ 3 \ 4) \downarrow 4
\end{align*}
\]
Question: could you write this proof..? What would happen if you tried...?

???

(\text{if (plus 1 \ - \ 1) 3 4 \downarrow 3})
\[
\text{If}_T : \frac{e_0 \downarrow n \quad n \neq 0 \quad e_1 \downarrow n'}{(\text{if } e_0 e_1 e_2) \downarrow n'} \\
\text{If}_F : \frac{e_0 \downarrow 0 \quad e_2 \downarrow n'}{(\text{if } e_0 e_1 e_2) \downarrow n'} \\
\square
\]

:(

(if (plus 1 \(-\) 1) 3 4) \downarrow 3

Answer: you \textbf{can't} write this proof, because \text{If}T will only let you evaluate \(e_1\) when \(e_0\) is non-0!
(plus (plus 0 1) 2) \downarrow 3

(plus (plus 0 1) 2) \downarrow 3

\( (\text{const} : \quad c \in \mathbb{Q}) \quad \text{plus} : \quad e_0 \downarrow n_0 \quad e_1 \downarrow n_1 \quad n' = n_0 + n_1 \)

\( (\text{div} e_0 \ e_1) \downarrow n' \)

\( (\text{div} e_0 \ e_1) \downarrow n' \)

\( (\text{not} \ e) \downarrow 1 \)

\( (\text{not} \ e) \downarrow 0 \)

\( (\text{if e}_0 \ e_1 \ e_2) \downarrow n' \)

\( (\text{if e}_0 \ e_1 \ e_2) \downarrow n' \)