

CS61A Discussion 9: **Tail Calls and Interpreters**

TA: **Jerry Chen**

Email: **jerry.c@berkeley.edu**

TA Website: **jerryjrchen.com/cs61a**

Attendance

Form: **tinyurl.com/jerrydisc**

For the weekly question,

- **please complete the quiz AND**
- **what would you like to see in disc?**

(Of course, please only check in if you showed up!)

Agenda

1. Week in Review
2. Halting Problem (fun diversion)
3. Tail Calls
4. Interpreters

Week In Review

MT2 — how was it?

- Regrades are open, please review the rubric and submit if applicable

Lab 10 (Interpreters) - **Due Friday**

Hw6 - **Due Friday**

Proj2 - Due 4/23

Maps Composition - Resubmit by next Friday

Halting Problem ("fun" diversion)

An **interpreter** is a program that **understands other programs**

Great, we can write **programs that analyze other programs!**

Are there any limitations to what we can calculate?

Halting Problem

The Halting Problem:

`halts?(P, x) :`

`return HALTS if P(x) will halt`

`return LOOP if P(x) will loop forever`

Halting Problem

```
DEFINE DOESITHALT(PROGRAM):  
{  
    RETURN TRUE;  
}
```

THE BIG PICTURE SOLUTION
TO THE HALTING PROBLEM

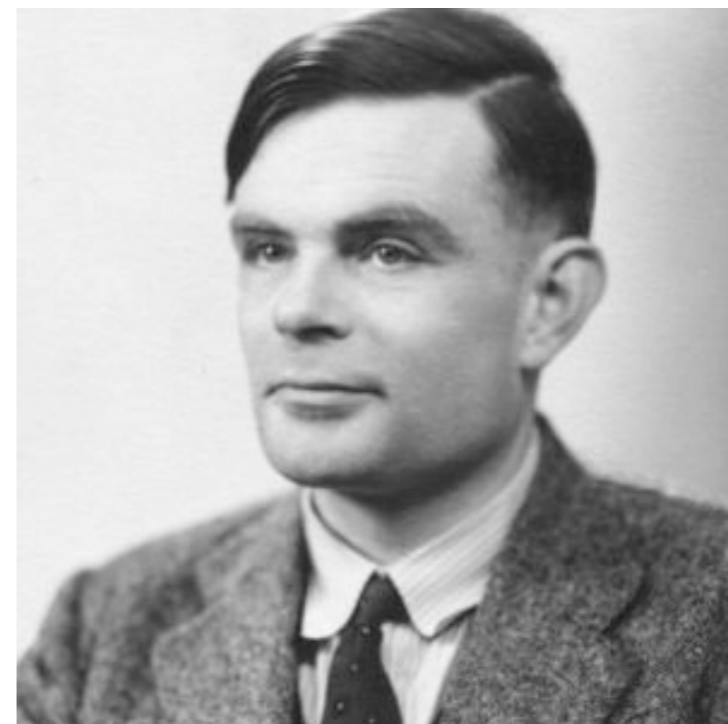
<https://xkcd.com/1266/>

Halting Problem

The program halts? cannot exist!

- First shown by Alan Turing

There are many other such problems that can be proved to be **uncomputable**



Halting Problem

```
trouble(P) :
```

```
    LOOP forever if halt?(P, P) == HALTS
```

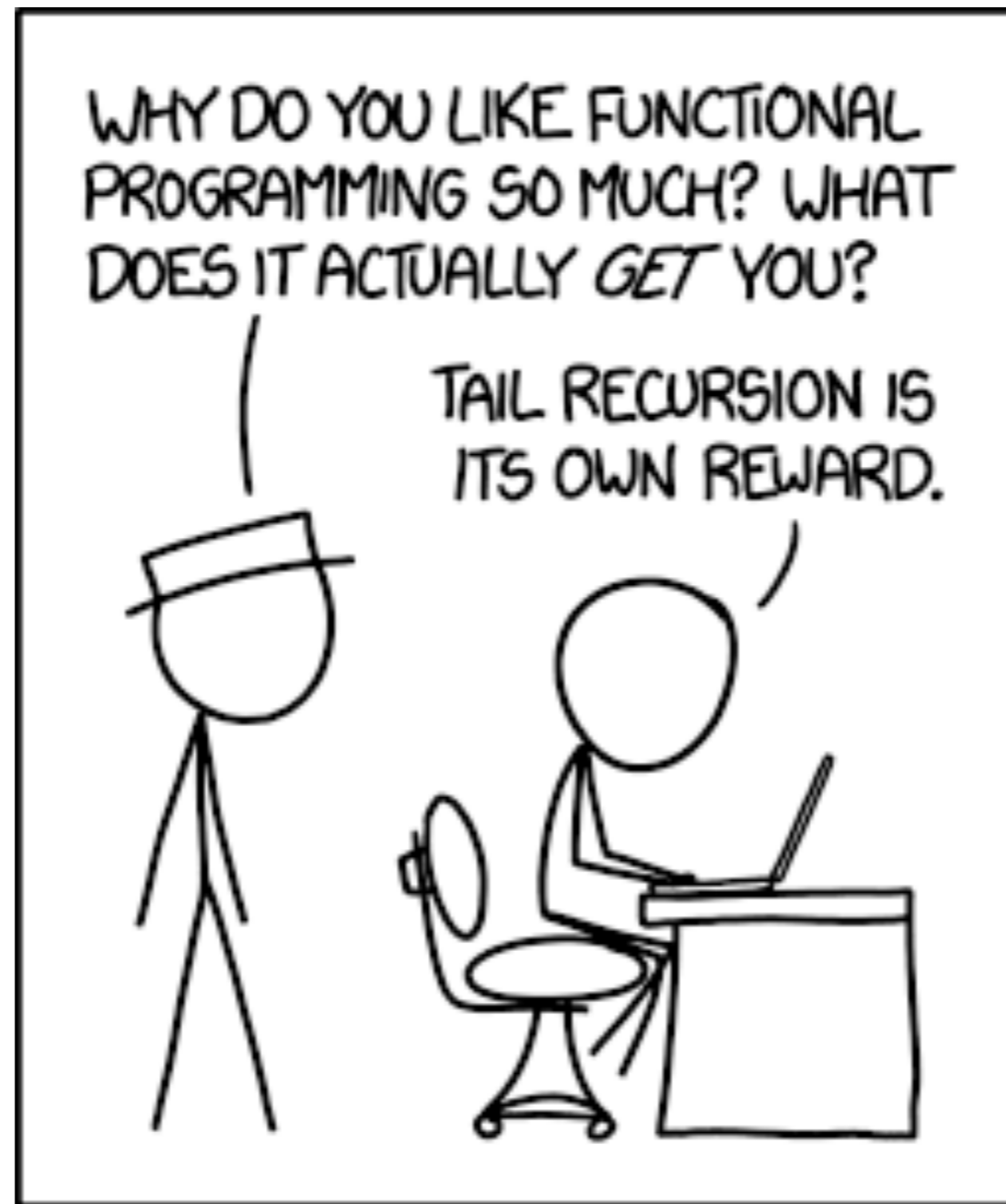
```
    else exit the function
```

What will `halt?(trouble, trouble)` return?

Either way, it's a contradiction!

More of this topic (computability) in CS 70, CS 170,
CS 172

Tail Calls



<http://xkcd.com/1270/>

Tail Calls

Scheme is **recursion only**

- Usually, recursive calls will take up space (think extra frames in the env diagram)
- **Tail calls** allow **recursion using constant space**
=> efficiency of iteration!
- **Tail recursion** is recursive calls performed at the end ("tail") of a function

Tail Calls

Big idea: with a valid tail call setup, **a recursive call** does not need anything **from the current frame** after it returns

- Put another way, after we do the recursive call, we **do not need to return for any computation**
- This is important because it means we can **reuse the current frame!** (might still need info for lookups)

Tail Calls

```
(define (fact n)  
  (if (= n 0)  
    1  
    (* n (fact (- n 1)))))
```

Tail Calls

```
(define (fact n)
```

```
  (define (fact-tail n result)
```

```
    (if (= n 0)
```

```
      result
```

```
      (fact-tail (- n 1) (* n result))))
```

```
(fact-tail n 1))
```

Tail Calls

Usually use a **helper** function to **track state**

Recursive call must be in a **tail context** to be a valid tail call

Tail Context

Tail contexts are essentially places we know a function terminates from ("tail end")

There's a list of them in the discussion handout. Think about why they make sense!

Tail Calls

Summary

- **Tail calls let us use constant space** for recursive calls
- To do a tail call, must perform **recursive calls in a valid tail context**
- Valid tail contexts are at certain "tails" of expressions, and **must not require addl. work** after the recursive call

Tail Recursion



<http://i.stack.imgur.com/qRlvz.jpg>

Calculator

The humble **Calculator** language:

- Polish-prefix notation
- Math only
- (Scheme... but less impressive)

Calculator

Supports argument nesting, and the 4 basic arithmetic operations:

```
> (+ (* 4 500) (- 26 (/ 20 2)))
```

2016

Calculator

Expressions are Pairs... seem familiar?

Calculator expressions structured (mostly) the same as Scheme expressions

Pair is the Python data structure equivalent for Scheme cons

Calculator

Recall: evaluating call expressions

- **Evaluate** the operator
- **Evaluate** the operands
- **Apply** the operator to the operands