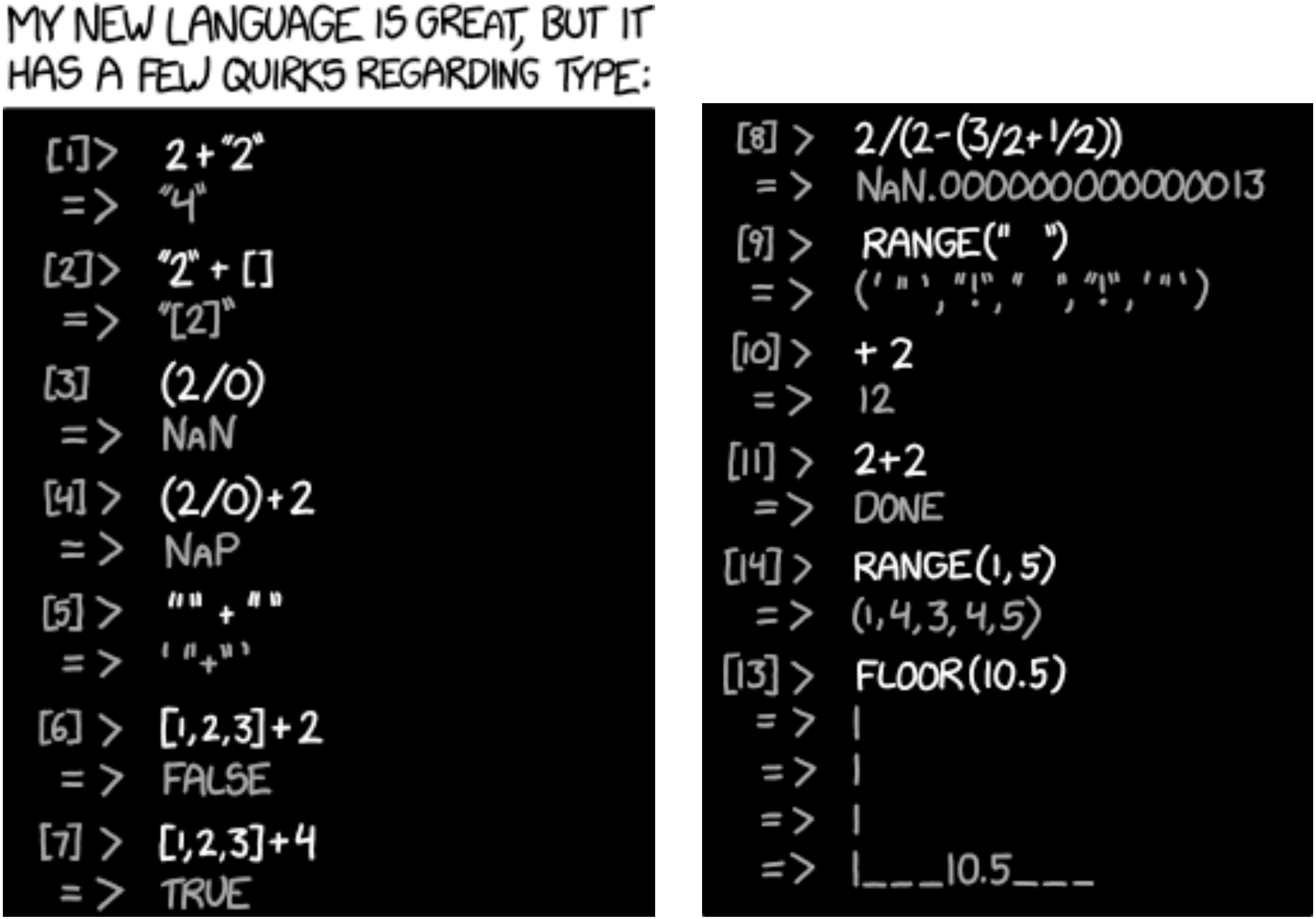
MY NEW LANGUAGE IS GREAT, BUT IT HAS A FEW QUIRKS REGARDING TYPE:
[1]> 2+"2"
=> "4"
[2]> "2" + []
=> "[2]"
[3] (2/0)
=> NaN
[4]> (2/0)+2
=> NaP
[5]> "" + ""
=> '"+"'
[6]> [1,2,3]+2
=> FALSE
[7]> [1,2,3]+4
=> TRUE
[8]> 2/(2-(3/2+1/2))
=> NaN.000000000000013
[9]> RANGE(" ")
=> ('"', "!", " ", "!", '"')
[10]> +2
=> 12
[11]> 2+2
=> DONE
[14]> RANGE(1,5)
=> (1,4,3,4,5)
[13]> FLOOR(10.5)
=> |
=> |
=> |
=> |___10.5___

For more insanity:

https://www.destroyallsoftware.com/talks/wat

[Broader point: No one (few people) knows what their programs do in untyped languages.]

Type Checking Basics

COS 326
David Walker
Princeton University

Logistics

Sign up for Piazza, our Q&A forum: http://piazza.com

Assignment #1 is due on Wednesday at 11:59pm

Last Time

Functional programming history

- Church & the lambda calculus
- Scheme
- ML
- Modern times: F#, Clojure, Scala, Map-Reduce, ...

What is functional programming?

- Imperative languages get most work done by *modifying* data
- Functional languages get most work done by analyzing old data and producing *new*, *immutable* data

OCaml

- Simple, typed programming language based on the lambda calculus
- Immutable data is the default; mutable data is possible

Type Checking

- Every value has a type and so does every expression
- We write (e : t) to say that expression e has type t. eg:

2 : int	"hello" : string
2 + 2 : int	"I say " ^ "hello" : string

Type Checking Rules

- There are a set of simple rules that govern type checking
  - programs that do not follow the rules will not type check and O'Caml will refuse to compile them for you (the nerve!)
  - at first you may find this to be a pain …
- But types are a great thing:
  - they *help us think* about *how to construct* our programs
  - they help us *find stupid programming errors*
  - they help us track down compatibility errors quickly when we edit and *maintain our code*
  - they allow us to *enforce powerful invariants* about our data structures

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int. (By rule 1)

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)
5 : int (By rule 1)

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for a … ")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

… tring_of_int e : string

FYI: This is a ***formal proof*** that the expression is well-typed!

- Using the rules:

2 : int and 3 : int.	(By rule 1)
Therefore, (2 + 3) : int	(By rule 3)
5 : int	(By rule 1)
Therefore, (2 + 3) * 5 : int	(By rule 4 and our previous work)

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Another perspective:

???? * ???? : int

rule (4) for typing expressions
says I can put any expression
with type int in place of the ????

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Another perspective:

7 * ???? : int

rule (4) for typing expressions
says I can put any expression
with type int in place of the ????

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Another perspective:

7 * (add_one 17) : int

rule (4) for typing expressions says I can put any expression with type int in place of the ????

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
         Objective Caml Version 3.12.0
#
```

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
```

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
#
```

press return and you find out the type and the value

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# "hello " ^ "world";;
- : string = "hello world"
#
```

press return and you find out the type and the value

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
      Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# “hello ” ^ “world”;;
- : string = “hello world”
# #quit;;
$
```

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Violating the rules:

"hello" : string (By rule 2)
1 : int (By rule 1)
1 + "hello" : ?? (NO TYPE! Rule 3 does not apply!)

Type Checking Rules

- Violating the rules:

```
# "hello" + 1;;
Error: This expression has type string but an
expression was expected of type int
```

- The type error message tells you the type that was expected and the type that it inferred for your subexpression
- By the way, this was one of the nonsensical expressions that did not evaluate to a value
- It is a ***good thing*** that this expression does not type check!

"Well typed programs do not go wrong"

Robin Milner, 1978

Type Checking Rules

- Violating the rules:

```
# "hello" + 1;;
Error: This expression has type string but an
expression was expected of type int
```

- A possible fix:

```
# "hello" ^ (string_of_int 1);;
- : string = "hello1"
```

- *One of the keys to becoming a good ML programmer is to understand type error messages.*

Example Type-checking Rules

if e1 : bool
and e2 : t and e3 : t (the same type t, for some type t)
then if e1 then e2 else e3 : t (that same type t)

Type Checking Rules

- Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

Type Checking Rules

- Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

Type Checking Rules

- Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

- Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

- Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

they don't *agree*!

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

???

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

- Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

they don't *agree*!

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

The type checker points to the correct branch as the cause of an error because it does not AGREE with the type of an earlier branch. Really, the error is in the earlier branch.

Moral: *Sometimes you need to look in an earlier branch for the error* even though the type checker points to a later branch. The type checker doesn't know what the user wants.

A Tactic: Add Typing Annotations

```
let rec concatn (s:string) (n:int) : string =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

```
Error: This expression has type int but an
expression was expected of type string
```

EXCEPTIONS:
DO THEY CAUSE PROGRAMS TO "GO WRONG"?

Type Checking Rules

- What about this expression:

```
# 3 / 0 ;;
Exception: Division_by_zero.
```

- Why doesn't the ML type checker do us the favor of telling us the expression will raise an exception?

Type Checking Rules

- What about this expression:

```
# 3 / 0 ;;
Exception: Division_by_zero.
```

- Why doesn't the ML type checker do us the favor of telling us the expression will raise an exception?
  - In general, detecting a divide-by-zero error requires we know that the divisor evaluates to 0.
  - In general, deciding whether the divisor evaluates to 0 requires solving the halting problem:

```
# 3 / (if turing_machine_halts m then 0 else 1);;
```

- There are type systems that will rule out divide-by-zero errors, but they require programmers supply proofs to the type checker

Isn't that cheating?

"*Well typed programs do not go wrong*"

Robin Milner, 1978

(3 / 0)   is well typed.   Does it "go wrong?" Answer: No.

"Go wrong" is a technical term meaning, "have no defined semantics." Raising an exception is perfectly well defined semantics, which we can reason about, which we can handle in ML with an exception handler.

So, it's not cheating.

(Discussion: why do we make this distinction, anyway?)

Type Soundness

"Well typed programs do not go wrong"

Programming languages with this property have *sound* type systems. They are called *safe* languages.

Safe languages are generally *immune* to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc. (but not immune to all bugs!)

Safe languages: ML, Java, Python, …

Unsafe languages: C, C++, Pascal

Well typed programs do not go wrong

Robin Milner

Turing Award, 1991

"For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics."

"Well typed programs do not go wrong"

Robin Milner, 1978

SUMMARY

OCaml

OCaml is a *typed* programming language

- the *type* of an expression *correctly predicts* the kind of *value* the expression will generate when it is executed
- there are systematic rules defining when any expression (or program) type checks
  - these rules actually for a formal logic ... it is not a coincidence that languages like ML were used inside theorem provers ... more later
- the type system is *sound*; the language is *safe*