Sealion

Bonnie Ishiguro & Ian Hill

Programming Languages Fall 2016 Final Project

Sealion

What is it?

A modified version of the language for homework 7

What were we doing?

Compiling Sealion into C

What's different?

Sealion is partially typed

```
int global_number = 10;
def int main () {
  int x = global_number;
  print x;
  print x + 2;
  print x * 2;
  print x + 2 * 2;
  print x * 2 + 2;
  print (x - 2) + 2;
  print (x + 2) * 2;
  print x * 2 + x * 2;
}
```

Sealion

What is it?

A modified version of the language for homework 7

What were we doing?

Compiling Sealion into C

What's different?

Sealion is partially typed

```
#include <stdio.h>
int global_number = 10;
int main () {
  int x = (global_number);
  printf("%i\n", (x));
  printf("%i\n", ((x)+2));
  printf("%i\n", ((x)*2));
  printf("%i\n", ((x)+(2*2)));
  printf("%i\n", ((x)*(2+2)));
  printf("%i\n", (((x)-2)+2));
  printf("%i\n", (((x)+2)*2));
  printf("%i\n", ((x)*(2+((x)*2))));
}
```

Our journey

Continuation Passing Style (CPS)

- Program uses less of the stack
- CPS would have generated much more complex C code

Type-checking

- We thought it would be necessary in order to create a typed language
- We hate our programmers, so we let the compiler type check instead

Typing

- Typing or type inference was required to compile a partially typed language into a strongly typed language

Types

C is strongly typed

C print statements require flags that indicate argument type

printf(“%s”, apple);   // %s → string

Type classes inspired by Lecture 10: “Notes on Static Types”

Type checking for primitive operations

```
class TString (Type):
    def __init__ (self):
        self.type = "string"
    def __str__ (self):
        return "string"
    def isString (self):
        return True
    def isEqual (self,t):
        return (t.isString() or
        t.isAny())
```

Compiling into C

- Root compile function accepts an .sl file and writes to an out.c file
- Compile method for each expression class

```
def compile (self, env):
    # compiling logic
    return (lines, output_type, anons)
```

Lines: compiled C code for each line of SL code

Output type: return type of evaluated expression

Anons: anonymous function declarations required for execution

Compiling into C

Compile function for EValue

```
def compile (self, env):
    return ([str(self._value)], self._value.type, [])
```

Compile function for EFunction

```
def compile (self, env):
    params = ",".join(["{} {}".format(str(t),p) for (t, p) in self._params])
    (lines, output_type, anons) = self._body.compile(env)
    return (["{} {} ({}) {{ {} }}".format(
        str(self._output_type),
        self._name,
        params,
        "".join(lines)
      )], self._output_type, anons)
```

Demo

Reflection

Does it work?

Well… yes. So far, the **Sealion compiler works** for simple code.

What would be next?

— We almost implemented **type inference** as part of implementing EPrint.

— We built infrastructure to handle **anonymous functions** but didn’t get around to fully supporting them.