Caml tail

Jennifer Lam
Fiona Rowan
Sandra Shaefer
Serena Shah-Simpson
Motivation

Want to do an interesting project

Don’t know anything about Ocaml

Let’s make an Ocaml-like language!
Our Goal & Challenges

Create a functional language that handles assignments, functions, and scoping in a similar way to Ocaml.

Compile to C.

Challenges:

Functional -> Imperative
Compiler Architecture

Source Code (*.tail) → Scanner → Parser → AST

Codegen → Transformer → SAST → Analyzer (Semantic Checker)

Executable (*.c)
Language Features

### Types
- Int | Float | String | Bool | Unit | Lists

### Operators
- +    -    *    /    
- <    >    <=   >=   ==
- &&   ||    not

### Control Flow
- if (a == b) then
  let bool c = true
else
  let bool c = false
;
match a with
  b -> print_int(1)
  | c -> print_int(2)
  | _ -> print_int(0)
;

### Builtins
- print_int(3);
- print_float(3.14);
- print_string(“tail”);
- print_bool(true);
Basic Language Parsing

Tail is made up of a list of **statements** separated by **semicolons**.

Every **statement** is essentially one of 3 types:

1. function assignment
2. variable assignment
3. expression
Expression Statements

Expressions do the bulk of the work in tail.

An *expression* statement goes to one of 4 things:

1. exprs (Literals, Binops, Uniops, Calls, Prepend )
2. Definitions (function or variable definitions within another expression)
3. Conditionals ( IF expr THEN expr ELSE expr)
4. match expressions (MATCH expr WITH match_list)
Assignment Statements

**Variable Assignments**

*Expression Definition*
let int a = {int literal};

*List Definition*
let int [] a = [ {literal list} ];

**Function Assignments**

let int a (int b, string c) = fun
{expression list} ;
Transformations

The **analyzer** goes through the AST for type checking and basic scope checking.

The **transformer** takes the SAST and produces a flattened, imperative version.

**Codegen** takes this and produces the C code.
Demo
Demo.tail

```c
#include <stdio.h>
#include <string.h>

int c = 2;
int a = 4;
int b = 2;
int d = 4 / 2;

int add(int q, int s) {
    int y = (q) + (s);
    return y;
}

int matchDemo (int b) {
    int a = add(b, c);
    switch(a){
    case(1):
        printf("%d\n", 1);
        break;
    case(2):
        printf("%d\n", 2);
        break;
    default:
        printf("%s", "wildcard");
    }
    return 0;
}

int main(){
    printf("comment -> expected output, nothing\n");
    printf("\n");
    printf("matchDemo(0) -> expected output, 2\n");
    matchDemo(0);
    printf("\n");
    printf("matchDemo(-1) -> expected output, 1\n");
    matchDemo(-1);
    printf("\n");
    printf("matchDemo(4) -> expected output, wildcard\n");
    matchDemo(4);
    printf("\n");
    printf("\n");
    let int a = 4;
    let int b = 2;
    printf("\n");
    printf("\n");
    printf("\n");
}```
Output of demo.tail

```
jennifer@plt-sandwich:/compiler$ ./tail < demo.tail > output.c; gcc output.c; ./a.out
  comment -> expected output, nothing
matchDemo(0) -> expected output, 2
    2
matchDemo(-1) -> expected output, 1
    1
matchDemo(4) -> expected output, wildcard
  wildcard
  d = a/b where a is 4 and b is 2 -> expected output, 2
  d =
    2
jennifer@plt-sandwich:/compiler$
```
Hi, this is a funny demo.

```c
#include <stdio.h>
#include <string.h>

int people = 4;
int sleep = 120;
int total = 120 + 4;
int bugs = 20000;

int add(int q, int s) {
    int y = q + s
    return y;
}

int matchDemo (int b) {
    int mix = add(b, bugs);
    switch(mix){
        case(0) :
            print_string("Output = tall");
            break;
        case(20124) :
            print_string("Output = tall");
            break;
        default :
            print_string("wildcard");
    }
}

int main(){
    printf("Number of people = 4\n");
    printf("Number of hours of sleep lost (conservative estimate) = 120\n");
    printf("When you add those all together and match on mix, you get: \n");
    matchDemo(total);
    print_string("Number of things learned = uncountable\n");
}
```
Output of demo2.c

```
sandra@sandra-VirtualBox:~/Documents/compiler$ ./demo2
Number of people = 4
Number of hours of sleep lost (conservative estimate) = 120
Number of bugs = 20000
When you add those all together and match on mix, you get:
Output = tail
Number of things learned = uncountable
```
Testing

- Wrote tests for features that we implement to make sure they work as expected
  - Tests are either expected to compile and run (test-*.tail files) or fail to compile (fail-*.tail files)
- Wrote a bash script that inputs *.tail files to our compiler and outputs a test*.c file, and then evaluates the following shell commands:
  - gcc -o test* test*.c
  - (and if the above works) ./test*
- The test script outputs these evaluations, and reports how many of the files output the expected output
Workflow

1. Weekly Meetings
2. Git and Bitbucket
3. Group programming
Lessons Learned

- Starting early isn’t enough - make sure you keep up momentum throughout the semester!
- Make sure you know what you’re getting into before making important life (and project) decisions.
- OCaml is beautiful.
- You can’t learn LLVM in a night.
- Management is essential