# Writing a functional language compiler

Andrew Craig

2019/11/25

# What I did

Wrote a compiler in C and Assembly

Compiles simple functional language 'LFL' to Assembly

# PART I

1. Me
2. Language overview
3. Introduction to Compilers
4. Introduction to Assembly

# PART II

1. Compiling anonymous functions
2. Compiling closures

# Me

- Andrew Craig
- Data Scientist
- GitHub: @andycraig
- Twitter: @andrew_cb2

| When | Where | What |
|---|---|---|
| School | Australia | HyperCard |
| University | Australia | Haskell, Java |
| Work | Japan | Visual Basic |
| Research | Australia | Matlab |
| Research | UK | R, Python |
| Work | Japan | R, Python |
| Hobby | Japan | Clojure? Rust?? APL??? |

No Assembly!

# What I did

`example.code`:

```
(let inc (λ x (plus x 1))
    (inc 1))
```

```
$ bin/compile example.code example.asm # I wrote bin/compile
```

`example.asm`:

```
; Assembly code generated by compiler
    global main
    extern printf, malloc                   ; C functions
    extern make_closure, call_closure       ; built-in functions
    extern plus, minus, equals              ; standard library functions

    section .text
_f0:
    push rbp
    mov rbp, rsp
    sub rsp, 40         ; memory for local variables
    mov rax, rdi        ; pointer to vector of arguments
    mov rbx, QWORD [rax+0]     ; move x from heap
    mov QWORD [rbp-8], rbx     ; move x to stack
    mov rax, 1                 ; integer constant
```

# What I did

```
    mov QWORD [rbp-16], rax    ; preparing operand 2/2
    mov rax, QWORD [rbp-8]    ; access x
    mov QWORD [rbp-24], rax    ; preparing operand 1/2
    mov rax, plus             ; access plus
    mov QWORD [rbp-32], rax    ; preparing closure
    mov rdx, QWORD [rbp-16]    ; operand 2/2
    mov rsi, QWORD [rbp-24]    ; operand 1/2
    mov rdi, QWORD [rbp-32]    ; operator location
    call call_closure            ; output goes to rax
    leave
    ret

main:
    push rbp
    mov rbp, rsp
    sub rsp, 32        ; memory for local variables
    mov rdx, 0            ; number of free variables
    mov rsi, 1            ; number of bound variables
    mov rdi, _f0            ; name of function
    call make_closure
    mov QWORD [rbp-8], rax    ; let inc
    mov rax, 1                ; integer constant
    mov QWORD [rbp-16], rax    ; preparing operand 1/1
    mov rax, QWORD [rbp-8]    ; access inc
    mov QWORD [rbp-24], rax    ; preparing closure
    mov rsi, QWORD [rbp-16]    ; operand 1/1
```

# What I did

```
    mov rdi, QWORD [rbp-24]    ; operator location
    call call_closure            ; output goes to rax
    mov rsi, rax        ; will print rax
    mov rdi, message
    mov rax, 0
    call printf
    mov rax, 0            ; exit code 0
    leave
    ret

    section .data
message: db "%d", 10, 0        ; 10 is newline, 0 is end-of-string
```

# What I did

## Convert Assembly versions

```
$ nasm -f elf64 example.asm
```

## Link

```
$ gcc -no-pie -o example example.o lib/libclosure.a lib/libstandard.a
```

## Run executable

```
$ ./example
2
```

# What I did

```
$ cat example.asm
```

```
...
_f0:
    push rbp
    mov rbp, rsp
    sub rsp, 40         ; memory for local variables
    mov rax, rdi        ; pointer to vector of arguments
...
```

```
$ objdump -d example
```

```
...
0000000000400540 <_f0>:
  400540:    55                     push   %rbp
  400541:    48 89 e5               mov    %rsp,%rbp
  400544:    48 83 ec 28            sub    $0x28,%rsp
  400548:    48 89 f8               mov    %rdi,%rax
  40054b:    48 8b 18               mov    (%rax),%rbx
  40054e:    48 89 5d f8            mov    %rbx,-0x8(%rbp)
  400552:    b8 01 00 00 00         mov    $0x1,%eax
  400557:    48 89 45 f0            mov    %rax,-0x10(%rbp)
  40055b:    48 8b 45 f8            mov    -0x8(%rbp),%rax
  40055f:    48 89 45 e8            mov    %rax,-0x18(%rbp)
  400563:    48 b8 3c 10 60 00 00   movabs $0x60103c,%rax
...
```

# Language (LFL)

```
(let f (lambda x (if (equals x 0)
                     0
                     (plus x 1)))
```

## Special forms

- let (letrec/def/defrec)
- lambda
- if

## Standard library

- plus
- minus
- equals

# Language (LFL)

## Types

- functions
- integers (8-byte)

## Use cases

- ?

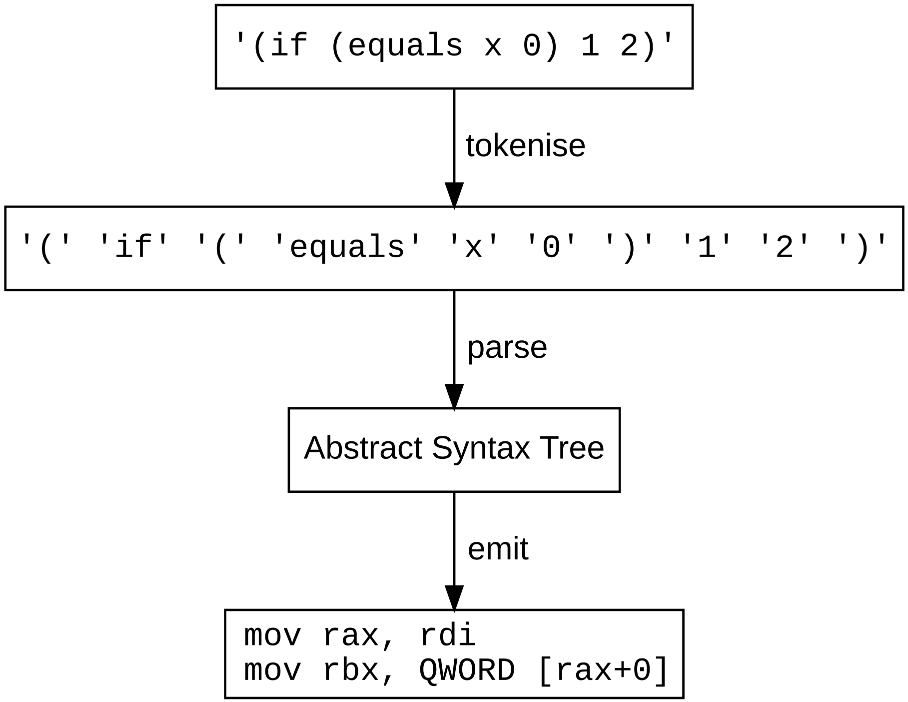
# Language (LFL)

## Functional?

- Anonymous functions ✓
- Closures ✓
- First-class functions ✓
- Recursion ✓
- Immutability ✓
- Purity ✓
- Pattern matching ×
- Macros ×
- Monads ×

# Compilers

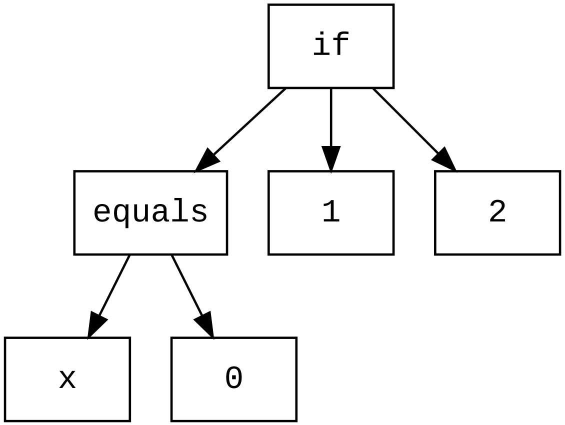
# Compilers


'(if (equals x 0) 1 2)'
tokenise
'(' 'if' '(' 'equals' 'x' '0' ')' '1' '2' ')'
parse
Abstract Syntax Tree
emit
mov rax, rdi
mov rbx, QWORD [rax+0]

# Compilers

```
(if (equals x 0) 1 2)
```

## Abstract Syntax Tree



# Assembly

```python
def inc(x):
    return x + 1

print(f"Hello World {inc(5)}")
```

```python
def inc(x):
    return x + 1

print(f"Hello World {inc(5)}")
```

```nasm
bits 64
     extern malloc, printf
     global    main

     section   .text
inc:                         ; function definition
     push rbp
     mov rbp, rsp
     mov rax, rdi         ; arguments: rdi, rsi, rdx, rcx, r8, r9
     add rax, 1           ; rax = rax + 1
     leave
     ret                  ; return rax
main:
     push rbp
     mov rbp, rsp
     mov rdi, 5           ; with argument 5,
     call inc             ; call function inc
     mov rsi, rax
     mov rdi, message
     mov rax, 0
     call printf
     xor rax, rax
     leave
     ret

     section   .data
message:  db        "Hello, World %ld", 10, 0  ; note the newline at the end
```

# Assembly

- Very 'global'
- 'Memory addresses', not 'variables'
- Everything is basically 'goto':
  - function call
  - if
  - loop

Many limits!

# Part II

# Main challenges

1. Anonymous functions
2. Closures

# Anonymous functions

Python:

```
lambda x: x + 1

map(lambda x: x + 1, [3, 4]) # [4, 5]

(lambda x: x + 1).__call__(3) # 4
```

# Anonymous functions

```
((lambda x (plus x 1)) 3)
```

```
(lambda x: x + 1).__call__(3) # 4
```

Q: Why are they difficult in Assembly?

```
main:
    push rbp
    mov rbp, rsp
    ???
    ??? 3
    call ???
    mov rax, 0
    leave
    ret
```

# Anonymous functions

```
((lambda x (plus x 1)) 3)
```

```
(lambda x: x + 1).__call__(3) # 4
```

Q: Why are they difficult in Assembly?

```
main:
    push rbp
    mov rbp, rsp
    anonymous:
        push rbp
        mov rbp, rsp
        mov rax, rdi
        add rax, 1
        leave
        ret
    mov rdi, 3
    call anonymous
    mov rax, 0
    leave
    ret
```

Segfault!

# Anonymous functions

## Problem

Function definitions must be 'global'

```
(lambda x: x + 1).__call__(3) # 4
```

## Solution

*Lambda lifting*: 'Lift' anonymous functions to global

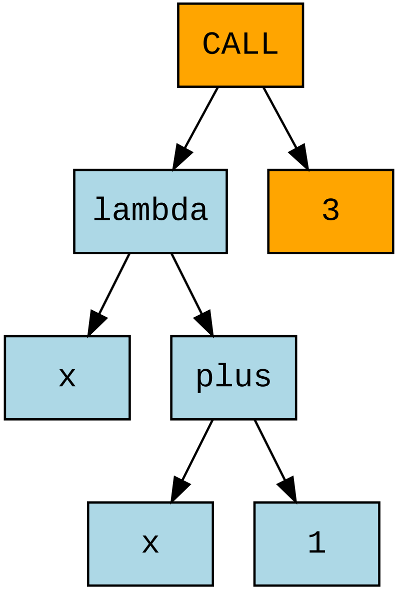
```
def _f0(x):
    return x + 1

_f0(3)
```

# Anonymous functions

1. Tokenise
2. Parse
3. Make Abstract Syntax Tree (AST)
4. Manipulate AST **(Lambda lifting)**
5. Output Assembly

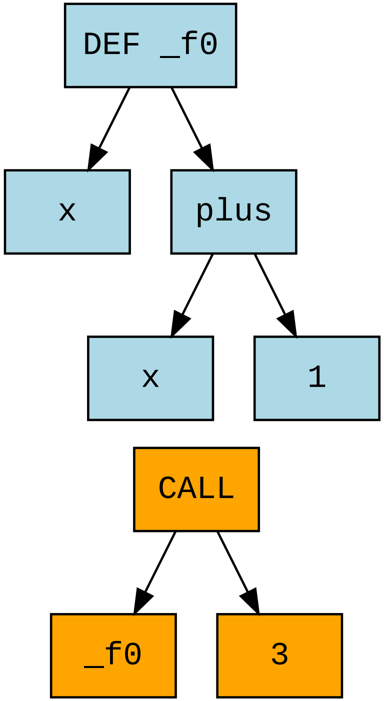
# Anonymous functions

```
((lambda x (plus x 1)) 3)
```



# Anonymous functions

```
((lambda x (plus x 1)) 3)
```



# Anonymous functions

```
...
_f0:
     push rbp
     mov rbp, rsp
     mov rax, rdi
     add rax, 1
     leave
     ret
main:
...
     mov rdi, 3
     call _f0
...
```

In emitted Assembly, no anonymous functions!

# Closures

# Closures

Python:

```
def make_adder(x):
    f = lambda y: y + x  # Anonymous function, closure over x
    return f

add_3 = make_adder(3)
add_5 = make_adder(5)

add_3(1) # 4
add_5(1) # 6
```

# Closures

Q: Why are they difficult in Assembly?

```
_f0:
    ; x + y
    ...

make_adder:
    push  rbp
    mov   rbp, rsp
    ??? _f0
    leave
    ret

main:
    push  rbp
    mov   rbp, rsp
    ??? 3
    call make_adder
    ??? ; RESULT
    ??? 5
    call make_adder
    ??? ; RESULT
    ; Use make_adder 3
    call ???
    pop   rbp
    leave
    ret
```

# Closures

Q: Why are they difficult in Assembly?

A: Because it's not clear where to 'save' the variables

# Closures

## A solution: *closure conversion*

```
def make_adder(x):
    f = lambda y: y + x  # _f0
    return f
```

'Closure struct'

```
MEMORY ADDRESS 1: 1 ('I am a closure')
MEMORY ADDRESS 2: _f0 (function name)
MEMORY ADDRESS 3: 1 (number of bound variables: y)
MEMORY ADDRESS 4: 1 (number of free variables: x)
MEMORY ADDRESS 5: [value of x] (value of free variable)
```

# Closures

```
def make_adder(x):
    f = lambda y: y + x  # _f0
    return f

add_3 = make_adder(3)
add_5 = make_adder(5)

add_3(1) # 4
add_5(1) # 6
```

add_3:

```
MEMORY ADDRESS 1: 1 ('I am a closure')
MEMORY ADDRESS 2: _f0
MEMORY ADDRESS 3: 1 (y)
MEMORY ADDRESS 4: 1 (x)
MEMORY ADDRESS 5: 3
```

add_5:

```
MEMORY ADDRESS 6: 1 ('I am a closure')
MEMORY ADDRESS 7: _f0
MEMORY ADDRESS 8: 1 (y)
MEMORY ADDRESS 9: 1 (x)
MEMORY ADDRESS 10: 5
```

# Closures

```python
def make_adder(x):
    f = lambda y: y + x  # _f0
    return f

add_3 = make_adder(3)
add_5 = make_adder(5)

add_3(1) # 4
add_5(1) # 6
```

```python
def _f0(a, b):
    return a + b

def make_adder(x):
    # f = lambda y: y + x
    f = make_closure(_f0, 1, 1, x)
    return f

add_3_closure = make_adder(3)
add_5_closure = make_adder(5)

# add_3(1) # 4
call_closure(add_3_closure, 1) # 4
# add_5(1) # 6
call_closure(add_5_closure, 1) # 6
```

# Closures

1. Tokenise
2. Parse
3. Make Abstract Syntax Tree (AST)
4. Manipulate AST **(Lambda lifting, closure conversion)**
5. Output Assembly

Compiler identifies bound variables, free variables

# Closures

`make_closure` and `call_closure` are special functions

Start of all Assembly output:

```
; Assembly code generated by compiler
    global main
    extern printf, malloc                    ; C functions
    extern make_closure, call_closure        ; built-in functions
    extern plus, minus, equals               ; standard library functions

    section .text
_f0:
    push rbp
    mov rbp, rsp
...
```

# Summary

- One way to compile anonymous functions: lambda lifting
- One way to compile closures: closure conversion

# Lessons

- Learnt more than I expected about low-level programming
- Respect for Assembly programmers!

# Next

- Write an interpreter? (C?)
- Write a virtual machine? (OCaml?)

# Code

https://github.com/andycraig/functional-compiler

# Resources

Compiler Explorer: https://godbolt.org/

Closure conversion: How to compile lambda:
http://matt.might.net/articles/closure-conversion/

Northeastern University CS 4410/6410: Compiler Design:
https://course.ccs.neu.edu/cs4410/lec_lambdas_notes.html