CSE 110A: Winter 2020

Fundamentals of Compiler Design I

Numbers, Unary Operations, Variables

Owen Arden
UC Santa Cruz

Based on course materials developed by Ranjit Jhala

Lets Write a Compiler!

Our goal is to write a compiler which is a function:

```
compiler :: SourceProgram -> TargetProgram
```

In CSE 110A, `TargetProgram` is going to be a binary executable.

Lets write our first Compilers

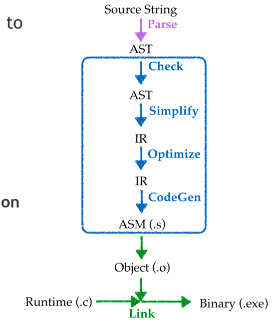
`SourceProgram` will be a sequence of *tiny* “languages”

- Numbers
  e.g. `7`, `12`, `42` …
- Numbers + Increment
  e.g. `add1(7)`, `add1(add1(12))`, …
- Numbers + Increment + Decrement
  e.g. `add1(7)`, `add1(add1(12))`, `sub1(add1(42))`
- Numbers + Increment + Decrement + Local Variables
  e.g. `let x = add1(7), y = add1(x) in add1(y)`

What does a Compiler look like?

An input source program is converted to an executable binary in many stages:

- **Parsed** into a data structure called an **Abstract Syntax Tree**
- **Checked** to make sure code is well-formed (and well-typed)
- **Simplified** into a convenient **Intermediate Representation**
- Optimized into (equivalent but) faster program
- **Generated** into assembly x86
- **Linked** against a run-time (usually written in C)



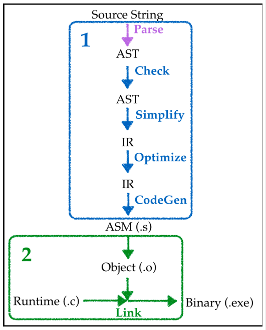
Compiler Pipeline

Simplified Pipeline

Goal: Compile *source* into *executable* that, when run, **prints** the result of evaluating the source.

Approach: Lets figure out how to write

- A **compiler** from the input *string* into *assembly*,
- A **run-time** that will let us do the printing.



Next, lets see how to do (1) and (2) using our sequence of `adder` languages.

Adder-1

Numbers

e.g. 7, 12, 42 …

The “Run-time”

Lets work *backwards* and start with the run-time.

Here’s what it looks like as a `C` program `main.c`

```
#include <stdio.h>

extern int our_code() asm("our_code_label");

int main(int argc, char** argv) {
  int result = our_code();
  printf("%d\n", result);
  return 0;
}
```

> `main` just calls `our_code` and prints its return value `our_code` is (to be) implemented in assembly.
>
> Starting at **label** `our_code_label` with the desired *return* value stored in register `EAX`, per the `C` calling convention

Test Systems in Isolation

Key idea in SW-Eng:

> *Decouple systems so you can test one component without (even implementing) another.*

Lets test our “run-time” without even building the compiler.

Testing the Runtime: A Really Simple Example

Given a `SourceProgram`

```
42
```

We *want to* compile the above into an assembly file `forty_two.s` that looks like:

```
section .text
global our_code_label
our_code_label:
  mov eax, 42
  ret
```

Testing the Runtime: A Really Simple Example

For now, lets just *write* that file by hand, and test to ensure object-generation and then linking works

```
$ nasm -f aout -o forty_two.o forty_two.s
$ clang -g -m32 -o forty_two.run forty_two.o main.c
```

On a Mac use `-f macho` instead of `-f aout`

We can now run it:

```
$ forty_two.run
42
```

Hooray!

The “Compiler”

First Step: Types

To go from source to assembly, we must do:

Our first step will be to **model** the problem domain using **types**.

The “Compiler”

Lets create types that represent each intermediate value:

- `Text` for the raw input source
- `Expr` for the AST
- `Asm` for the output x86 assembly

Defining the Types: Text

`Text` is raw strings, i.e. sequences of characters

```
texts :: [Text]
texts =
  [ "It was a dark and stormy night..."
  , "I wanna hold your hand..."
  , "12"
  ]
```

Defining the Types: Expr

We convert the `Text` into a tree-structure defined by the datatype

```
data Expr = Number Int
```

Note: As we add features to our language, we will keep adding cases to `Expr`.

Defining the Types: Asm

Lets also do this *gradually* as the x86 instruction set is HUGE!

Recall, we need to represent

```
section .text
global our_code_label
our_code_label:
  mov eax, 42
  ret
```

Defining the Types: Asm

An `Asm` program is a **list of instructions** each of which can:

- Create a `Label`, or
- Move a `Arg` into a `Register`
- `Return` back to the run-time.

```
type Asm = [Instruction]

data Instruction
  = ILabel Text
  | IMov   Arg Arg
  | IRet
```

Where we have

```
data Register
  = EAX

data Arg
  = Const Int -- a fixed number
  | Reg Register   -- a register
```

Second Step: Transforms

Ok, now we just need to write the functions:

```
-- 1. Transform source-string into AST
parse   :: Text -> Expr

-- 2. Transform AST into assembly
compile :: Expr -> Asm

-- 3. Transform assembly into output-string
asm     :: Asm  -> Text
```

Second Step: Transforms

Pretty straightforward:

```
parse :: Text -> Expr
parse    = parseWith expr
  where
    expr = integer

compile :: Expr -> Asm
compile (Number n) =
  [ IMov (Reg EAX) (Const n)
  , IRet
  ]

asm :: Asm -> Text
asm is = L.intercalate
     "\n" [instr i | i <- is]
```

Where `instr` is a `Text` representation of *each* `Instruction`

```
instr :: Instruction -> Text
instr (IMov a1 a2) =
  printf "mov %s, %s"
       (arg a1) (arg a2)

arg :: Arg -> Text
arg (Const n) = printf "%d" n
arg (Reg r)   = reg r

reg :: Register -> Text
reg EAX = "eax"
```

Brief digression: Typeclasses

Note that above we have *four* separate functions that crunch different types to the `Text` representation of x86 assembly:

```
asm   :: Asm -> Text
instr :: Instruction -> Text
arg   :: Arg -> Text
reg   :: Register -> Text
```

Remembering names is *hard*.

We can write an **overloaded** function, and let the compiler figure out the correct implementation from the type, using **Typeclasses**.

The following defines an *interface* for all those types `a` that can be converted to x86 assembly:

```
class ToX86 a where
  asm :: a -> Text
```

Brief digression: Typeclasses

Now, to overload, we say that each of the types `Asm`, `Instruction`, `Arg` and `Register` *implements* or **has an instance of** `ToX86`

```
instance ToX86 Asm where
  asm is = L.intercalate "\n" [asm i | i <- is]

instance ToX86 Instruction where
  asm (IMov a1 a2) = printf "mov %s, %s" (asm a1) (asm a2)

instance ToX86 Arg where
  asm (Const n) = printf "%d" n
  arg (Reg r)   = asm r

instance ToX86 Register where
  asm EAX = “eax"
```

Note in each case above, the compiler figures out the *correct* implementation, from the types…

Adder-2

Well that was easy! Lets beef up the language!

- Numbers + Increment
- e.g. `add1(7)`, `add1(add1(12))`, …

Repeat our Recipe

- Build intuition with **examples**,
- Model problem with **types**,
- Implement compiler via **type-transforming-functions**,
- Validate compiler via **tests**.

Example 1

How should we compile?

`add1(7)`

In English

- Move `7` into the `eax` register
- Add `1` to the contents of `eax`

In ASM

```
mov eax, 7
add eax, 1
```

Aha, note that `add` is a new kind of `Instruction`

Example 2

How should we compile

`add1(add1(12))`

In English

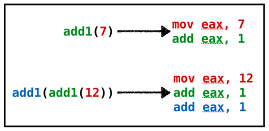
- Move `12` into the `eax` register
- Add `1` to the contents of `eax`
- Add `1` to the contents of `eax`

In ASM

```
mov eax, 12
add eax, 1
add eax, 1
```

Compositional Code Generation

Note correspondence between sub-expressions of source and assembly



```
add1(7) → mov eax, 7
          add eax, 1

add1(add1(12)) → mov eax, 12
                 add eax, 1
                 add eax, 1
```

We will write compiler in **compositional** manner

- Generating `Asm` for each *sub-expression* (AST subtree) independently,
- Generating `Asm` for *super-expression*, assuming the value of sub-expression is in `EAX`

Extend Type for Source and Assembly

Source Expressions

```
data Expr = ...
          | Add1   Expr
```

Assembly Instructions

```
data Instruction
  = ...
  | IAdd  Arg Arg
```

Examples Revisited

```
src1 = "add1(7)"

exp1 = Add1 (Number 7)

asm1 = [ IMov (EAX) (Const 7)
       , IAdd (EAX) (Const 1)
       ]

src2 = "add1(add1(12))"

exp2 = Add1 (Add1 (Number 12))

asm2 = [ IMov (EAX) (Const 12)
       , IAdd (EAX) (Const 1)
       , IAdd (EAX) (Const 1)
       ]
```

Transforms

Now lets go back and suitably extend the transforms:

```
-- 1. Transform source-string into AST
parse   :: Text -> Expr

-- 2. Transform AST into assembly
compile :: Expr -> Asm

-- 3. Transform assembly into output-string
asm     :: Asm  -> Text
```

Lets do the easy bits first, namely `parse` and `asm`

Parse

```
parse :: Text -> Expr
parse    = parseWith expr

expr :: Parser Expr
expr =  try primExpr
    <|> integer

primExpr :: Parser Expr
primExpr = Add1 <$> rWord "add1" *> parens expr
```

Asm

To update `asm` just need to handle case for `IAdd`

```
instance ToX86 Instruction where
  asm (IMov a1 a2) = printf "mov %s, %s" (asm a1) (asm a2)
  asm (IAdd a1 a2) = printf "add %s, %s" (asm a1) (asm a2)
```

Note

- GHC will *tell* you exactly which functions need to be extended (Types, FTW!)
- We will not discuss `parse` and `asm` any more…

Compile

Finally, the key step is

```
compile :: Expr -> Asm
compile (Number n)
  = [ IMov (Reg EAX) (Const n)
    ]
compile (Add1 e)
    -- EAX holds value of result of `e` ...
  = compile e
    -- ... so just increment it.
    ++ [ IAdd (Reg EAX) (Const 1) ]
```

Examples Revisited

Lets check that compile behaves as desired:

```
ghci> (compile (Number 12)
[ IMov (Reg EAX) (Const 12) ]

ghci> compile (Add1 (Number 12))
[ IMov (Reg EAX) (Const 12)
, IAdd (Reg EAX) (Const 1)
]

ghci> compile (Add1 (Add1 (Number 12)))
[ IMov (Reg EAX) (Const 12)
, IAdd (Reg EAX) (Const 1)
, IAdd (Reg EAX) (Const 1)
]
```

Adder-3

You do it!

- Numbers + Increment + Double
- e.g. `add1(7)`, `twice(add1(12))`, `twice(twice(add1(42)))`

Adder-4

- Numbers + Increment + Decrement + Local Variables
- e.g. `let x = add1(7), y = add1(x) in add1(y)`

Local variables make things more interesting

Repeat our Recipe

- Build intuition with **examples**,
- Model problem with **types**,
- Implement compiler via **type-transforming-functions**,
- Validate compiler via **tests**.

Examples

Example: let1

```
let x = 10
in
    x
```

Need to store 1 variable - `x`

Example: let2

```
let x = 10        -- x = 10
  , y = add1(x)   -- y = 11
  , z = add1(y)   -- z = 12
in
    add1(z)       -- 13
```

Need to store 3 variable - `x, y, z`

Example: let3

```
let a = 10
  , c = let b = add1(a)
        in
           add1(b)
in
    add1(c)
```

Need to store 3 variables - `a`, `b`, `c` - but at most 2 at a time

- First `a, b`, then `a, c`
- Don't need `b` and `c` simultaneously

Registers are Not Enough

A single register `eax` is useless:

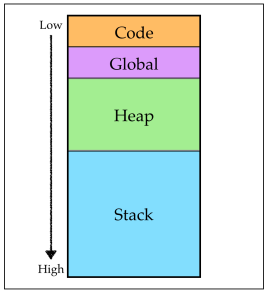
- May need 2 or 3 or 4 or 5 … values.

There is only a *fixed* number (say, `N`) of registers:

- And our programs may need to store more than `N` values, so
- Need to dig for more storage space!

Memory: Code, Globals, Heap and Stack

Here's what the memory - i.e. storage - looks like:



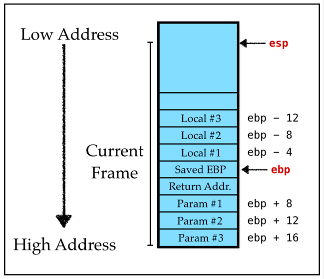
Focusing on “The Stack”

Lets zoom into the stack region, which when we start looks like this.

The stack **grows downward** (i.e. to **smaller** addresses)

We have lots of 4-byte slots on the stack at offsets from the “stack pointer” at addresses:

`[ESP - 4 * 1]`, `[ESP - 4 * 2]`, ...,



Mapping from variables to slots

The `i`-th *stack-variable* lives at address `[ESP - 4 * i]`

Required A mapping

- From *source variables* (`x`, `y`, `z` ...)
- To *stack positions* (`1`, `2`, `3` ...)

Solution The structure of the `let`s is stack-like too...

- Maintain an `Env` that maps `Id |-> StackPosition`
- `let x = e1 in e2` adds `x |-> i` to `Env`
  - where `i` is current height of stack.

Example: Let-bindings and Stacks

```
let x = 1              -- []
in                     -- [ x |-> 1 ]
  x
let x = 1              -- []
  , y = add1(x)        -- [x |-> 1]
  , z = add1(y)        -- [y |-> 2, x |-> 1]
in
    add1(z)            -- [z |- 3, y |-> 2, x |-> 1]
```

QUIZ

At what position on the stack do we store variable `c` ?

```
let a = 1
  , c =
      let b = add1(a)
      in add1(b)
in
   add1(c)
```

A. 1
B. 2
C. 3
D. 4
E. not on stack!

http://tiny.cc/cse110a-stackvar-ind

QUIZ

At what position on the stack do we store variable `c` ?

```
let a = 1
  , c =
      let b = add1(a)
      in add1(b)
in
   add1(c)
```

A. 1
B. 2
C. 3
D. 4
E. not on stack!

http://tiny.cc/cse110a-stackvar-grp

QUIZ

At what position on the stack do we store variable `c` ?

```
                           -- []
let a = 1                  -- [a |-> 1]
  , c =                    -- [a |-> 1]
      let b = add1(a)      -- [b |-> 2, a |-> 1]
      in add1(b)           -- [a |-> 1]
                           -- [c |-> 2, a |-> 1]
in
   add1(c)
```

QUIZ

```
                 -- ENV(n)
let x = STUFF
                 -- [x |-> n+1, ENV(n)]
in OTHERSTUFF
                 -- ENV(n)
```

Strategy

At each point, we have `env` that maps (previously defined) `Id` to `StackPosition`

Variable Use

To compile `x` given `env`

- Move `[ESP - 4 * i]` into `eax`

(where `env` maps `x |-> i`)

Variable Definition

To compile `let x = e1 in e2` we

- Compile `e1` using `env` (i.e. resulting value will be stored in `eax`)
- Move `eax` into `[ESP - 4 * i]`
- Compile `e2` using `env'`

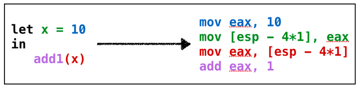
(where `env'` be `env` with `x |-> i` i.e. push `x` onto `env` at position `i`)

Example: Let-bindings to Asm

Lets see how our strategy works by example:

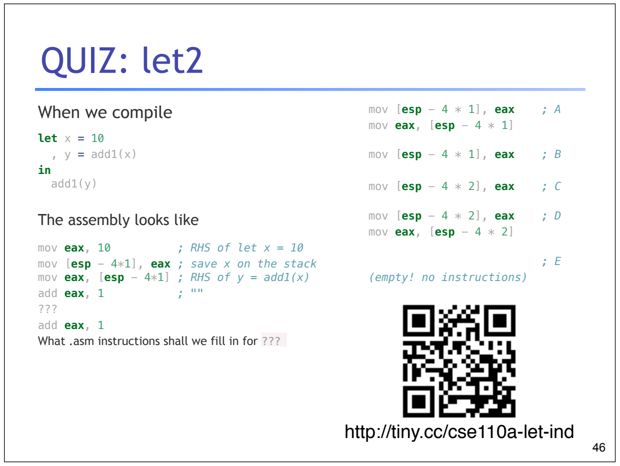
Example: let1

b



```
let x = 10
in
   add1(x)
```

```
mov eax, 10
mov [esp - 4*1], eax
mov eax, [esp - 4*1]
add eax, 1
```



QUIZ: let2

When we compile

```
let x = 10
  , y = add1(x)
in
  add1(y)
```

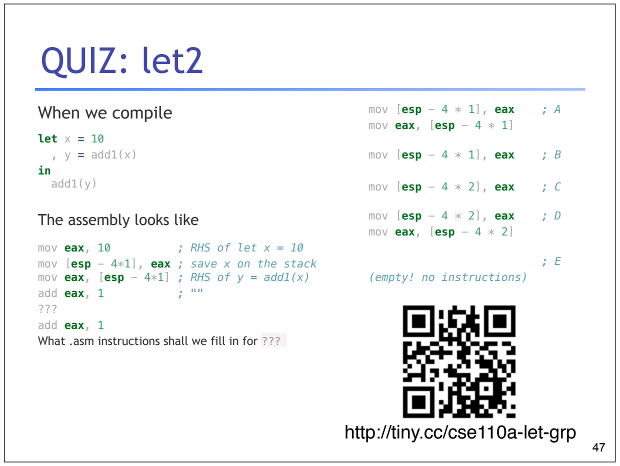
The assembly looks like

```
mov eax, 10           ; RHS of let x = 10
mov [esp - 4*1], eax ; save x on the stack
mov eax, [esp - 4*1] ; RHS of y = add1(x)
add eax, 1            ; ""
???
add eax, 1
```

What .asm instructions shall we fill in for `???`

```
mov [esp - 4 * 1], eax    ; A
mov eax, [esp - 4 * 1]

mov [esp - 4 * 1], eax    ; B

mov [esp - 4 * 2], eax    ; C

mov [esp - 4 * 2], eax    ; D
mov eax, [esp - 4 * 2]

                          ; E
(empty! no instructions)
```

http://tiny.cc/cse110a-let-ind



QUIZ: let2

When we compile

```
let x = 10
  , y = add1(x)
in
  add1(y)
```

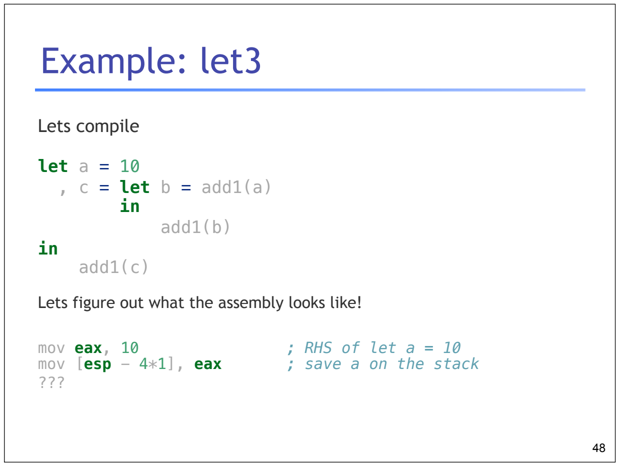
The assembly looks like

```
mov eax, 10           ; RHS of let x = 10
mov [esp - 4*1], eax ; save x on the stack
mov eax, [esp - 4*1] ; RHS of y = add1(x)
add eax, 1            ; ""
???
add eax, 1
```

What .asm instructions shall we fill in for `???`

```
mov [esp - 4 * 1], eax    ; A
mov eax, [esp - 4 * 1]

mov [esp - 4 * 1], eax    ; B

mov [esp - 4 * 2], eax    ; C

mov [esp - 4 * 2], eax    ; D
mov eax, [esp - 4 * 2]

                          ; E
(empty! no instructions)
```

http://tiny.cc/cse110a-let-grp



Example: let3

Lets compile

```
let a = let 10
  , c = let b = add1(a)
        in
            add1(b)
in
    add1(c)
```

Lets figure out what the assembly looks like!

```
mov eax, 10             ; RHS of let a = 10
mov [esp - 4*1], eax    ; save a on the stack
???
```

Types

Now, we're ready to move to the implementation!

Lets extend the types for *Source Expressions*

```
type Id   = Text

data Expr = ...
          -- `let x = e1 in e2` modeled as is `Let x e1 e2
          | Let Id Expr Expr    `
          | Var Id
```

Lets enrich the `Instruction` to include the register-offset `[esp - 4*i]`

```
data Arg = …
         -- `[esp - 4*i]` modeled as `RegOffset ESP i`
         | RegOffset Reg Int
```

Environments

Lets create a new `Env` type to track stack-positions of variables

```
data Env = [(Id, Int)]

data Maybe a = Nothing | Just a

lookupEnv :: Env -> Id -> Maybe Int
lookupEnv [] x                = Nothing
lookupEnv ((y, n) : rest) x = if x == y
                                then Just n
                                else lookupEnv rest x

pushEnv   :: Env -> Id -> (Int, Env)
pushEnv env x = (xn , env')
  where
    env'      = (x, xn) : env
    xn        = 1 + length env
```

Environments

```
compile env (Let x e1 e2) =
  compile env e1
  ++ -- EAX hold the value of "x"
  [IMov (RegOffset ESP xn) EAX ]
  ++
  compile env' e2
  where
    (xn, env') = pushEnv env x

compile env (Var x) = [IMov EAX (RegOffset ESP xn)]
  where
    xn              = case lookupEnv env x of
                        Just n -> n
                        Nothing -> error "variable out of scope"
```

Environments

API:

- **Push** variable onto `Env` (returning its position),
- **Lookup** variable's position in `Env`

```
push :: Id -> Env -> (Int, Env)
push x env = (i, (x, i) : env)
  where
    i = 1 + length env

lookup :: Id -> Env -> Maybe Int
lookup x []                = Nothing
lookup x ((y, i) : env)
  | x == y                 = Just i
  | otherwise              = lookup x env
```

Questions?