EECS 483 Lecture 3

Let-bindings and simple stack allocations

September 11, 2023

Recap

So far, our language was pretty simple:

> ‹*expr*›: ***NUMBER***

…with abstract syntax

```
type expr = int64
```

… and the compiler simply generated a mov instruction to place the integer into RAX

Refactoring the Compiler

When given a number, say 483, we generate the following assembly:

```
section .text
global start_here
start_here:
  mov RAX, 483
  ret
```

Only this line corresponds to our input program! The others are scaffolding.

Growing the language

Things to consider when we add a new feature:

1. Its impact on the *concrete syntax* of the language
2. Examples using the new enhancements, so we build intuition of them
3. Its impact on the *abstract syntax* and *semantics* of the language
4. Any new or changed *transformations* needed to process the new forms
5. Executable *tests* to confirm the enhancement works as intended

Concrete Syntax

‹expr›:

| *NUMBER*

| `add1` `(` ‹expr› `)`

| `sub1` `(` ‹expr› `)`

Examples

Concrete Syntax	Answer
`42`	`42`
`add1(42)`	`43`
`sub1(42)`	`41`
`sub1(add1(add1(42)))`	`43`

Abstract Syntax

```
pub enum Exp {
    Num(i64),
    Add1(Box<Exp>),
    Sub1(Box<Exp>),
}
```

Semantics: evaluate argument to a number, then add or subtract one from it

Transformations

New assembly instruction:

```
add <dest>, <val>
```

Increment the destination by the right-side value

Transformations (continued)

New definition of Instr:

```
enum Instr {
  ...
  Add(Reg, i32) /* Increment the left-hand reg by the
value of the right-hand immediate */
  // In x86 only 32-bit literals can be on the right side
of an add instruction
}
```

Example: compiling `add1(42)`

Two steps:

1. Load 42 into RAX
2. Add 1 to RAX

Resulting assembly:

```
mov RAX, 42
add RAX, 1
```

Another example

Compile `sub1(add1(add1(42)))`

How to handle subtraction? Just add -1.

```
mov RAX, 42
add RAX, 1
add RAX, 1
add RAX, -1
```

Notice that each piece of the program corresponds to a related piece of the assembly!

Important Observation: Compositionality

Our translations are **compositional**: a translation of a composite expression is just a function of the translations of its constituent parts!

This makes writing the compilation function easy; we can use recursion

Compile `add1(e)`

```
Instrs from compiling e…
add RAX, 1
```

Correctness

The specification is that the compiled code outputs the same answer. How do we **know** our compilation is always correct?

Compile `add1(e)`

```
Instrs for e…
add RAX, 1
```

Testing the Feature

After implementing the code for the feature, we should now test that it works as expected:

1. Unit tests: check that compile_to_instrs outputs the exact right sequence of instructions
2. Integration tests: check that the compiled program has the same output as the interpreter

Adding `let`

Growing the language

Things to consider when we add a new feature:

1. Its impact on the *concrete syntax* of the language
2. Examples using the new enhancements, so we build intuition of them
3. Its impact on the *abstract syntax* and *semantics* of the language
4. Any new or changed *transformations* needed to process the new forms
5. Executable *tests* to confirm the enhancement works as intended

Concrete Syntax for Let

⟨expr⟩: . . .

| ***IDENTIFIER***

| `let` ***IDENTIFIER*** `=` ⟨expr⟩ `in` ⟨expr⟩

Abstract Syntax for Let

```
enum Exp {
  ...
  Id(String),
  Let(String, Box<Exp>, Box<Exp>)
}
```

Concrete Syntax for Let

⟨*expr*⟩: . . .

| ***IDENTIFIER***

| **`let`** ***IDENTIFIER*** **`=`** ⟨expr⟩ **`in`** ⟨expr⟩

Discuss: Examples? What are the edge cases?

Examples

```
let x = 5 in add1(x)

    => 6

let x = 483 in (let y = add1(x) in add1(y))

    => 485

let x = (let y = add1(5) in add1(y)) in add1(x)

    => 8
```

Examples

```
let x = 5 in add1(x)

    => 6

let x = 483 in (let y = add1(x) in add1(y))

    => 485

let x = (let y = add1(5) in add1(y)) in add1(x)

    => 8
```

Examples

```
let x = 5 in add1(x)

=> add1(5)

=> 6
```

Examples

```
let x = 483 in (let y = add1(x) in add1(y))

=> let y = add1(483) in add1(y)

=> let y = 484 in add1(y)

=> add1(484)

=> 485
```

Semantics: Writing an Interpreter for the New Language

Same as before:

- Numbers evaluate to themselves
- Adding or subtracting one should evaluate the expression and then add/subtract one from the result

But what about identifiers and let-bindings?

Lazy vs Eager Evaluation

In **lazy** evaluation, an identifier is evaluated to a result on an as-needed basis.

In **eager** evaluation, an expression is fully evaluated before it is bounded to an identifier, and is subsequently never evaluated again.

Discussion:

1. When is **lazy** evaluation **more efficient**?
2. When is **eager** evaluation **more efficient**?
3. Do **lazy** and **eager** evaluation ever have **different results**?

Environments

We need to track the meaning of each identifier. We will do so using an **environment**.

Possible choices for the type of the environment:

- Match each identifier to the expression it was bound to
    - Environment type: `[(&str, Exp)]`
    - *Lazy* behavior
- Match each identifier to the result of evaluating that expression
    - Environment type: `[(&str, i64)]`
    - *Eager* behavior

Scope

Scope tells us which names are available for use within a given expression.

Our convention for scope: the program `let x = e1 in e2` means that x can be used in e2, but not in e1.

> Is this code valid?
>
> ```
> let x = add1(x) in x
> ```
>
> **No, because x is not in scope in add1(x)!**
>
> (If the language supported recursion, this kind of definition could be sensible, but even if it did, in this particular example, there would be no solution, i.e., no x such that x = x + 1.)

Important Convention

What is the result of the following code:

```
let x = 1 in let x = 2 in x
```

Important Convention

What is the result of the following code:

```
let x = 1 in let x = 2 in x
```

Choices: 1, 2, or error

Our convention: answer = 2

“Inner bindings shadow outer ones.”

Interpreter Demo

(Look at example Rust code)

The Stack

Compiling the New Language

How can we compile programs in our updated language?

- No notion of identifier names or environments in the assembly language
- One register is not enough, since we may need to track multiple names at once. (In fact, no fixed number of registers would be enough.)

Solution

Insight #1: Broaden our notion of a name

In the interpreter, a name was used to map to a value or expression.

In reality, any unique identifier will suffice, and all values will need to exist in memory at runtime.

So now, instead of “a name is a string”, we should think “a name is a memory address”.

Insight #2

While compiling, we can maintain an environment of type `Vec<&str, Address>`.

- When we compile a **let-binding**, we can extend this environment with new addresses for new identifiers.
- When we compile an **identifier**, we look up the relevant address.

This environment is not needed at runtime!

New question: how do we assign addresses to identifiers?

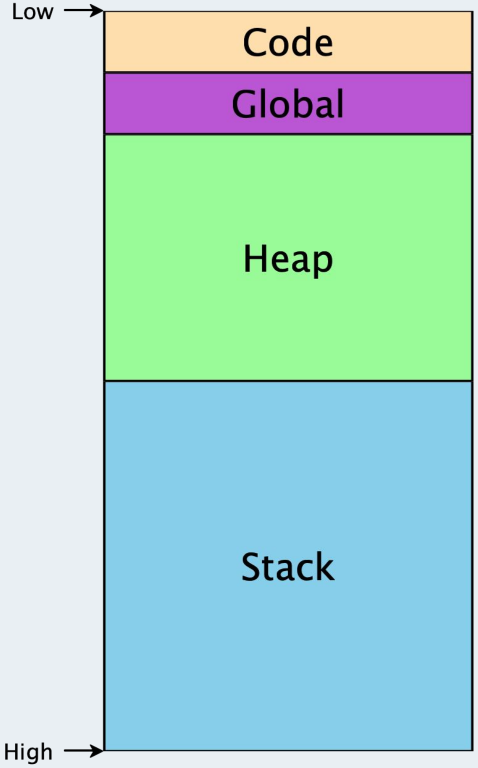
Memory Layout

Conceptually, memory is an array of bytes, addressed from 0 to 2^64 (assuming a 64-bit machine).

There are restrictions on what addresses can be used.

The typical memory layout for a program is shown on the next slide.

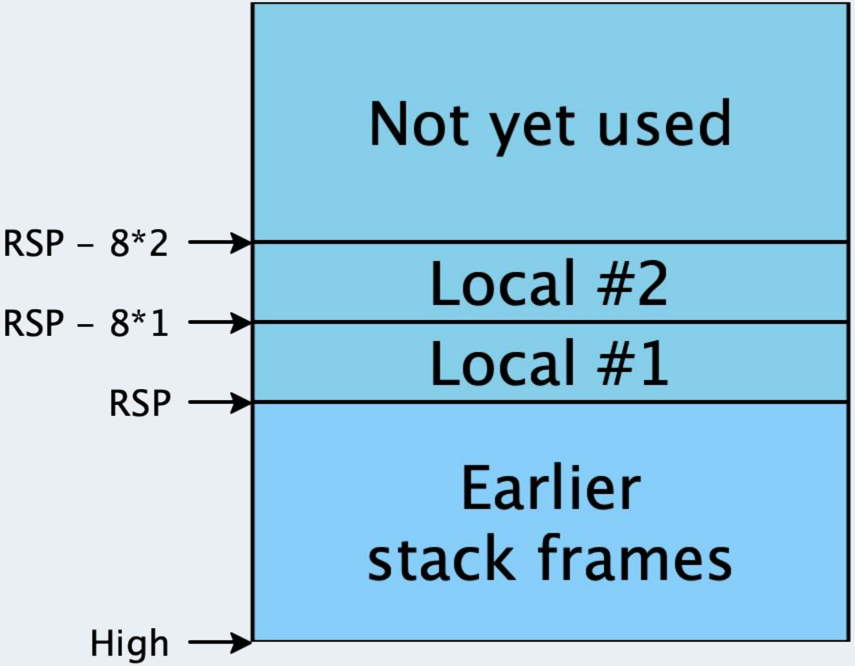
Memory Layout (continued)



Sections of Program Memory

- Code/text segment: includes the program machine code
- Global segment: global data available throughout the program’s execution
- Heap: memory that is dynamically allocated as the program runs
- **Stack: used as the program calls and returns from functions**

Stack Layout



Stack Layout (continued)

- The stack is divided into stack frames, with each function in progress gettings its own frame.
- Each stack frame can be used freely by its corresponding function.
- When the function returns, its stack frame is freed for use by future calls.
- The RSP register contains the address where the current stack frame begins.

Allocating Identifiers on the Stack

With the above knowledge, the task of assigning addresses is more concrete.

We are given addresses on the stack at `RSP - 8 * 1, RSP - 8 * 2, ... RSP - 8 * i`

We need to allocate a number to each identifier so that identifiers needed simultaneously are mapped to different numbers.

Naive Allocation Algorithm

Give every unique binding its own unique integer, i.e., every binder gets its own stack slot.

Implementation: keep a global mutable counter of the number of variables we have seen, and a global table mapping names to counterns.

Naive Allocation: Examples

```
  let x = 10          /* [] */
in add1(x)            /* [ x --> 1 ] */
```

```
   let x = 10          /* [] */
in let y = add1(x)     /* [x --> 1] */
in let z = add1(y)     /* [y --> 2, x --> 1] */
in add1(z)             /* [z --> 3, y --> 2, x --> 1] */
```

Naive Allocation: Examples

```
  let a = 10                            /* [] */
in let c =     let b = add1(a)          /* [a --> 1] */
            in let d = add1(b)          /* [b --> 2, a --> 1] */
            in add1(b)                  /* [d --> 3, b --> 2, a --> 1] */
in  add1(c)                             /* [c --> 4, d --> 3, b --> 2, a --> 1] */
```

Problems with this Approach

Wastes space (see last line of last example where neither b nor d are in scope, but their stack slots are still reserved).

Need to be careful when using **mutable state**!

Another Attempt

Observation: as we enter the bodies of let-expressions, only the bindings of those particular let-expressions are in scope; everything else is unavailable.

We can trace a straight-line path from any given let-body out through its parents to the outermost expression of a given program.

So, we only need to maintain uniqueness among the variables on those paths!

Example

The first two examples shown earlier are the same under this new strategy. Here is the last example:

```
  let a = 10                    /* [] */
in let c =    let b = add1(a)   /* [a --> 1] */
           in let d = add1(b)   /* [b --> 2, a --> 1] */
           in add1(b)           /* [d --> 3, b --> 2, a --> 1] */
in  add1(c)                     /* [c --> 2, a --> 1] */
```

Resulting Assembly Code

```
let a = 10

in let c =    let b = add1(a)

           in let d = add1(b)

           in add1(b)

in  add1(c)
```

```
mov RAX, 10
mov [RSP - 8*1], RAX
mov RAX, [RSP - 8*1]
add RAX, 1
mov [RSP - 8*2], RAX
mov RAX, [RSP - 8*2]
add RAX, 1
mov [RSP - 8*3], RAX
mov RAX, [RSP - 8*2]
add RAX, 1
mov [RSP - 8*2], RAX
mov RAX, [RSP - 8*2]
add RAX, 1
```