EECS 483: Compiler Construction

Lecture XX:
Conditionals 1

Month Day, 2025

Announcements

- Assignment 1 is due on Friday, the 31st.
- Next assignment to be released on Monday, February 3rd.

Questions from Last Lecture?

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?

2. What is the semantics of the language we are compiling?

3. How can we implement that semantics in assembly code?

4. How can we generate that assembly code programmatically?

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?
2. What is the semantics of the language we are compiling?
3. How can we implement that semantics in assembly code?
4. **How should we adapt our intermediate representation to new features?**
5. How can we generate assembly code from the IR?

Snake v0.2: "Boa"

In Adder we developed straightline code that performed arithmetic operations and stored variables and intermediate results in memory.

In Boa, we extend this to include conditional and looping control flow.

Snake v0.2: "Boa"

In Adder we developed straightline code that performed arithmetic operations and stored variables and intermediate results in memory.

In Boa, we extend this to include conditional and looping control flow.

Snake v0.2: "Boa"

⟨*expr*⟩: …

  | `if` ⟨expr⟩ `:` ⟨expr⟩ `else:` ⟨expr⟩

Abstract Syntax

```
enum Exp {
  ...
  If { cond: Box<Exp>, thn: Box<Exp>, els: Box<Exp> }
}
```

Examples, Semantics

We only have one datatype of integers, no separate booleans. We'll use C's convention: 0 is false and everything else is true

Concrete Syntax	Answer
`if 5: 6 else: 7`	6
`if 0: 6 else: 7`	7
`if sub1(1): 6 else: 7`	7

Examples, Semantics

Again we have added if as an **expression** form (like Rust), so we need to handle cases like

```
(if x: 6 else: 8) + (if y: x else: 3)
```

similar to C's ternary operator `x ? 6 : 8`

For this reason, if expressions always have an else branch

Examples, Semantics

We want to ensure that our if expressions only evaluate **one** of the two branches at runtime, and not both.

How would you test that you did this correctly? What kinds of programs would behave differently if you always evaluated both branches?

```
if x:
  print(1)
else:
  print(0)
```

```
let x = 1 in
if x:
  7
else:
  infinite-loop
```

Scope

How should scoping extend to if expressions?

Should the following program be considered well scoped?

```
def main(x):
if 0:
   y
else:
   x
```

Control Flow in x86

x86 Instruction Semantics

So far, instructions execute in sequence. Why?

The instruction to execute is determined by a special register, the **instruction pointer "rip"**.

in our abstract machine, each execution step starts by interpreting the memory at **[rip]** as a binary encoding of an assembly code instruction.

Most instructions (mov, add, etc) increment rip by the size of the encoded instruction, meaning at the next step the instruction pointer will execute the instruction after it in memory

What instruction have we seen so far that works differently?

x86 Instruction Semantics

So when we look at our code, we should think of it that we are looking at that code laid out in memory.

Assembly code **labels** give names to memory addresses.

```
entry:
   mov rax, rdi
   sub rax, 1
   cmp rax, 0
   je thn
els:
   mov rax, 7
   ret
thn:
   mov rax, 6
   ret
```

x86 Instructions: jmp

```
jmp loc
```

Semantics: sets the instruction pointer to loc.

Often loc is a **label** for another instruction in the same assembly file, but it doesn't have to be, it can be a register, or a memory location, or even a constant (almost certainly will crash in that case)

x86 Instructions: j*cc*

j*cc* loc

Actually a family of instructions, where **cc** is a **condition code**

Semantics: sets rip to loc **if the condition code is satisfied**, otherwise increment rip like a sequential instruction.

x86 RFLAGS

The x86 abstract machine includes a register **rflags**, which like **rip** is manipulated as a side-effect of many instructions.

rflags is a 64-bit register, each bit acting as a boolean flag. Most of these are irrelevant to our compiler (or unused). The most relevant to us are

- OF "overflow flag": 1 if an overflow occurs, otherwise 0

- SF "sign flag": 1 if the output is negative, otherwise 0

- ZF "zero flag": 1 if the output is zero, otherwise 0

x86 RFLAGS

The x86 abstract machine includes a register **rflags**, which like **rip** is manipulated as a side-effect of many instructions.

mov does not affect flags

add, sub, imul, other arithmetic expressions do:

```
mov rax, 15
mov rcx, 17
sub rax, rcx
```

```
OF: 0
SF: 1
ZF: 0

rax: -2
rcx: 17
```

x86 Instruction: cmp

Often we want to set **rflags**, but not actually store an arithmetic result:

```
cmp arg1, arg2
```

"compare instruction". Behaves like **sub** for the purposes of setting flags, but does **not** update arg1

```
mov rax, 15
mov rcx, 17
cmp rax, rcx
```

```
OF: 0
SF: 1
ZF: 0

rax: 15
rcx: 17
```

x86 Instruction: test

Often we want to set **rflags**, but not actually store an arithmetic result:

```
test arg1, arg2
```

"test instruction". Behaves like a bitwise **and** for the purposes of setting flags, but does **not** update arg1. Useful for checking certain bits are set

x86 Condition codes

Condition codes interpret the flags as a boolean formula. Mnemonic makes the most sense if we have just run a **sub** or **cmp** operation

- e (equal): ZF
- ne (not equal): ~ ZF
- l (less than): OF ^ SF
- le (lesser or equal): (OF ^ SF) | ZF
- g (greater than): ~ le = ~ ((OF ^ SF) | ZF)
- ge (greater or equal): ~ l = ~ (OF ^ SF)

x86 Instructions: j*cc*

```
jcc loc
```

Actually a family of instructions, where **cc** is a **condition code**

Semantics: sets rip to loc **if the condition code is satisfied**, otherwise increment rip like a sequential instruction.

```
je loc

jle loc

jg loc

...
```

x86 Conditional Control Flow: Example

```
def main(x):
  if sub1(x):
    6
  else:
    7
```

➡

```
entry:
  mov rax, rdi
  sub rax, 1
  cmp rax, 0
  jne thn
els:
  mov rax, 7
  ret
thn:
  mov rax, 6
  ret
```

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?
2. What is the semantics of the language we are compiling?
3. How can we implement that semantics in assembly code?
4. **How should we adapt our intermediate representation to new features?**
5. How can we generate assembly code from the IR?

SSA

Previously:

- one single block of operations ending in a return
- compiled to a block of sequential assembly labeled entry, ending in a ret

Extend as follows:

- add ability to define additional labeled blocks called **basic blocks**
- add ability to end a block by **branching** rather than returning

SSA Abstract Syntax

```
pub enum BlockBody {
    Terminator(Terminator),
    Operation {
        dest: VarName,
        op: Operation,
        next: Box<BlockBody>,
    },
    SubBlock {
        block: BasicBlock,
        next: Box<BlockBody>,
    },
}
```

```
pub struct BasicBlock {
    pub label: Label,
    pub body: BlockBody,
}
```

```
pub enum Terminator {
    Return(Immediate),
    ConditionalBranch {
        cond: Immediate,
        thn: Label,
        els: Label,
    },
}
```

SSA Concrete Syntax

```
entry(x):
  thn:
    ret 6
  els:
    ret 7
  sub1_arg = x
  cond = sub sub1_arg 1
  cbr cond thn els
```

Compiling Basic Blocks to x86

For each basic block, we will emit a block of assembly code with a label corresponding to the name of the block.

Need to ensure that the sub-blocks are emitted after the instructions for the current block.

Conditional branches can be encoded using a mix of x86 conditional jumps and unconditional jumps

Compiling Basic Blocks to x86

```
entry(x):
  thn:
    ret 6
  els:
    ret 7
  sub1_arg = x
  cond = sub sub1_arg 1
  cbr cond thn els
```

```
entry:
        mov [rsp + -8], rdi
        mov rax, [rsp + -8]
        mov [rsp + -16], rax
        mov rax, [rsp + -16]
        mov r10, 1
        sub rax, r10
        mov [rsp + -24], rax
        mov rax, [rsp + -24]
        cmp rax, 0
        jne thn#0
        jmp els#1
thn#0:
        mov rax, 6
        ret
els#1:
        mov rax, 7
        ret
```

Compiling Conditionals to (Sub-)blocks

```
def main(x):
  if sub1(x):
    6
  else:
    7
```

→

```
entry(x):
  thn:
    ret 6
  els:
    ret 7
  sub1_arg = x
  cond = sub sub1_arg 1
  cbr cond thn els
```

Conditionals and Continuations

```
enum Exp {
  ...
  If { cond: Box<Exp>, thn: Box<Exp>, els: Box<Exp> }
}
```

Strategy:

Make basic blocks for thn and els, giving them unique label names, compiling them recursively

Compile cond, do a conditional branch on the result, using the label names generated for thn and els

Compiling Conditionals to (Sub-)blocks

```
if cond:
  thn
else:
  els
```

➡

```
thn%uid:
  ... thn code
els%uid':
  ... els code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Conditionals and Continuations

This works if the result of the if expression is to be returned, but what if it's more complex:

```
let x = (if y: 5 else: 6) in
x * 3
```

We need to also account for the **continuation** of the if expression!

The continuation is what should happen **after** the result of the expression is computed. Now that result might be computed in either branch.

So the continuation needs to be run after either branch

Compiling Conditionals by Copying Continuations

```
if cond:
  thn
else:
  els
```

```
thn%uid:
  ... thn code
els%uid':
  ... els code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Copying Continuations

```
if cond:
   thn
else:
   els
```

+

```
... continuation code
```

→

```
thn%uid:
  ... thn code
  ... continuation code
els%uid':
  ... els code
  ... continuation code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Copying Continuations

```
let x = (if y: 5 else: 6) in
x * 3
```

➡

```
thn%0:
  x%2 = 5
  res%3 = x%2 * 3
  ret res%3
els%1:
  x%4 = 6
  res%3 = x%4 * 3
  ret res%3
cbr y%5 thn%0 els%1
```

Compiling Conditionals by Copying Continuations

Strategy:

Make basic blocks for thn and els, giving them unique label names, compiling them recursively

Compile cond, do a conditional branch on the result, using the label names generated for thn and els

For continuations: copy them into both branches

For next time:

The strategy we've described today does create "correct" code.

Why is the strategy completely infeasible in practice?