KU | Fall 2018 | Drew Davidson

EECS 665 COMPILER CONSTRUCTION

23 – Three Address Code

Homework 5

- Question 1:
  - The gist:

list -> list oneItem	`list.trans = list.trans U oneItem.trans`
…	
oneItem -> decl	`oneItem.trans = { }`
oneItem -> stmt	`oneItem.trans =  stmt.trans`
…	
stmt -> ID ASSIGN exp SEMI	`stmt.trans = {ID.value} U exp.trans`

Homework 5

- Question 2:
  - The gist: Annotate each parse tree node with its SDT value *according to the given SDT* goal

Homework 6

- Question 1:
  - The gist: add

    type -> ID

    typedef -> STRUCT ID ID SEMICOLON

Homework 6

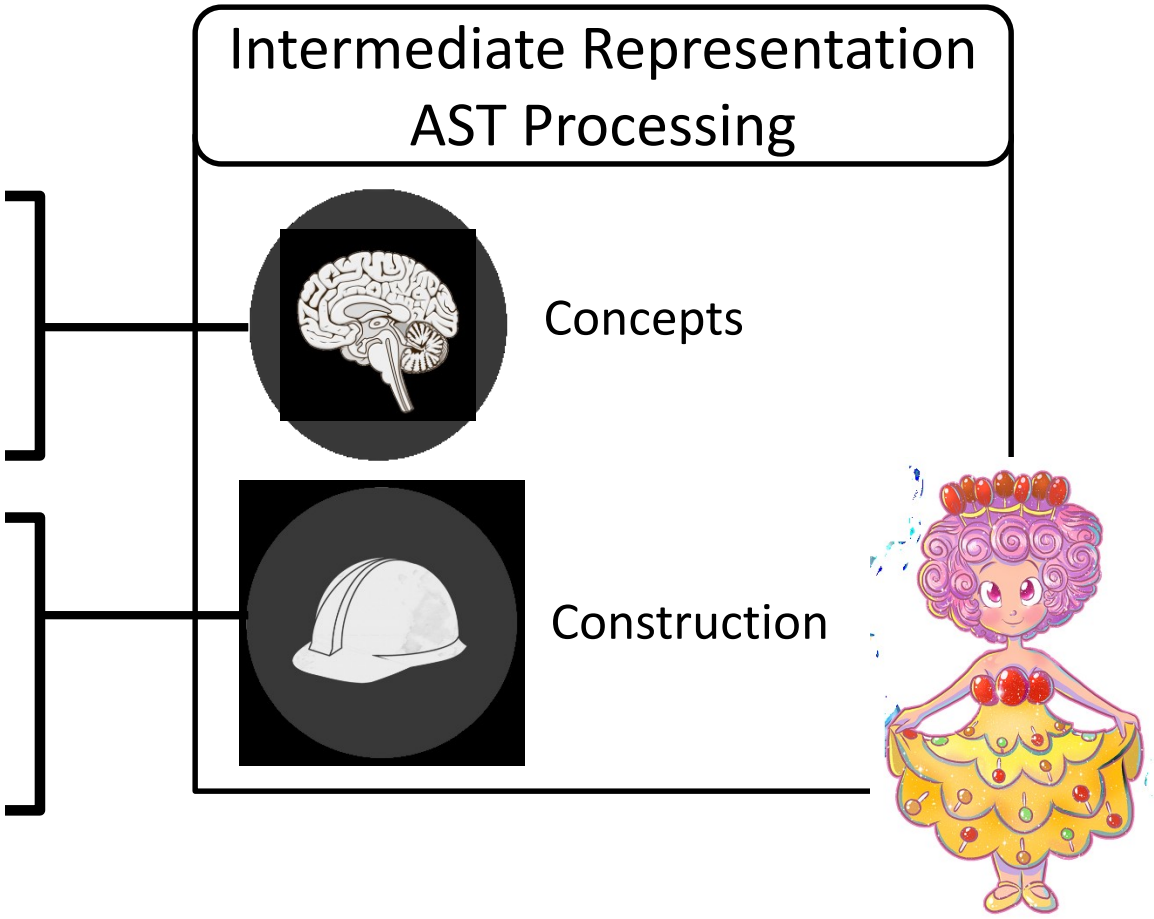
- Question 2:
  - A) Store name, type name, type value
  - B) Maybe the symbol table has a new kind for typedef, and stores a pointer to the “source type” of the typedef. On typedef use, recursively walk the typedef chain until you hit a base case
  - C) Store name, type, as you usually would, but if it’s not primitive you have to walk the tree to find the defn
  - D) look up the entry in the symbol table, potentially walk the typedef chain

Homework 6

- Question 2: Draw symbol table

Last Lecture

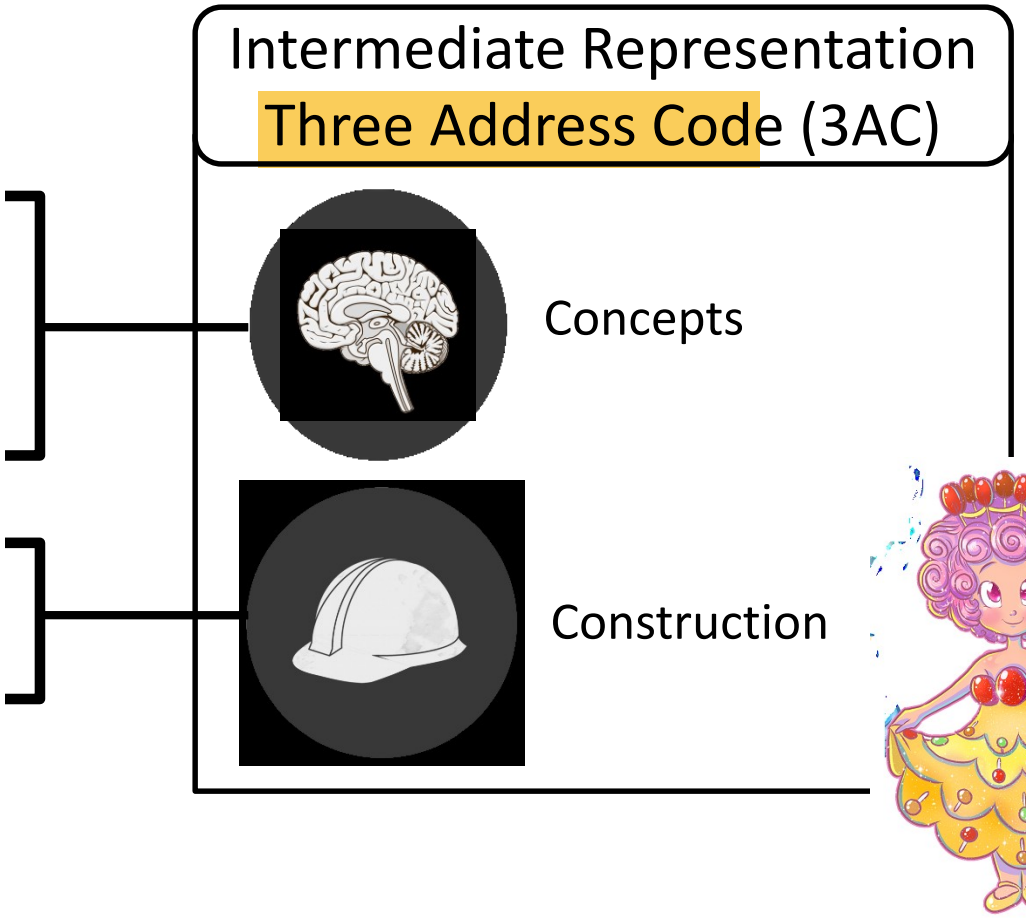
- What IRs are
- Why IRs are used
- Some IR forms
- AST Rewriting
  - Function inlining
  - Constant folding



This Lecture

- 3AC: Definition
- 3AC: Intuition
- 3AC: Examples

- From AST to 3AC



What 3AC is

An Instruction Set Architecture (ISA) designed for simplicity

- Family of pseudocode notations
- Linear representation of a program

3AC Key Features

- Simple model of memory
- Highly constrained set of instructions
- The most "complex" instruction has three operands and one operator (hence the name)

3AC: Operands

Data:

- For Lil' C, we only explicitly need variables
  - We'll assume constants and an infinite store of explicitly typed variables (we only need int for Lil' C)

examples

(We'll come back to the other operand categories later)

3AC: Assignment Operations

- Assignment forms
  - Binary:

$$X := Y \ \text{op} \ Z$$ `+ - * / % etc`

  - Unary:

X: = NOT Y
X:=X xor 1

$$X := \text{op} \ Y$$ **(simulate with binary)**

  - Assignment (Nullary):

x:=Y+0

$$X := Y$$ **(could simulate with binary)**

3AC Data Examples

- Translate the following into 3AC:

  EXAMPLE 1 (assume int decls)

```
a = b + c
d = a - 7
```

  EXAMPLE 2 (assume int decls)

```
a = -b + 1
```

  EXAMPLE 3 (assume bool decls)

```
a = true;
```

a:=1

3AC Data Exercises

- Use your intuition to translate to 3AC:

EXAMPLE 1 (assume

x = x * x

EXAMPLE 2

x = a + b * c

t1:= b* c
x:=a+t1

EXAMPLE 3

z = (a < (b + 1))

t1:= b+ 1
t2:=a<t1
z:=t2

Summary

3AC encodes most data operations as chains of assignments

3AC: Control Flow

Linear representation so…

- No nested: Text
  - Data expressions (a = b + c * d)
  - Control
    - (ifs / fors / whiles)
- We'll represent such structures using labels/jumps

3AC: Jumps

Label1: a:=5
b:= a+3
goto Label1

Labels:

Label: Instr

Unconditional:

GoTo Label

Conditional

IfZ Val GoTo label

IF zero go to label

3AC Jumps: Example

```
bool x = true;
int y = 1;
if (x) {
  y = 2;
}
```

How to jump

x:= 1;	1
y:=1;	2
ifz x Goto Label 1	3
y:=2;	4
Label1: line 5	5

3AC Jumps: Exercise

```
int x;
int y;
while (x < y) {
      x = x * 2;
}
y = x;
```

While loop is different ifz
Label2: t0 := x LT y
ifz t0 Goto label1

x::= x* 2;
Goto: label 2

Label1: y:=x;

Summary

3AC encodes intraprocedural control operations as jumps

3AC Function Calls

Calls are a bit more complicated

- How do we deal with locals & arguments?
- What about with recursion
  - Could extend the memory model to include a stack
  - Could encode the function semantics as instructions

3AC Function Calls

Caller instructions

```
set_arg x, k
call p
retrieve x
```

(x is the k^{th} param)

(Actually transfer control)

(*Get return value)*

3AC Function Calls

Caller instructions

```
set_arg x, k
call p
retrieve x
```

(x is the k^{th} param)

(Actually transfer control)

(*Get return value)*

Summary

3AC encodes intraprocedural control operations as jumps

3AC Function Calls

Some quick terminology

- Caller
  - The source function
- Callee
  - The destination function

CALLER

call

CALLEE

3AC Function Calls

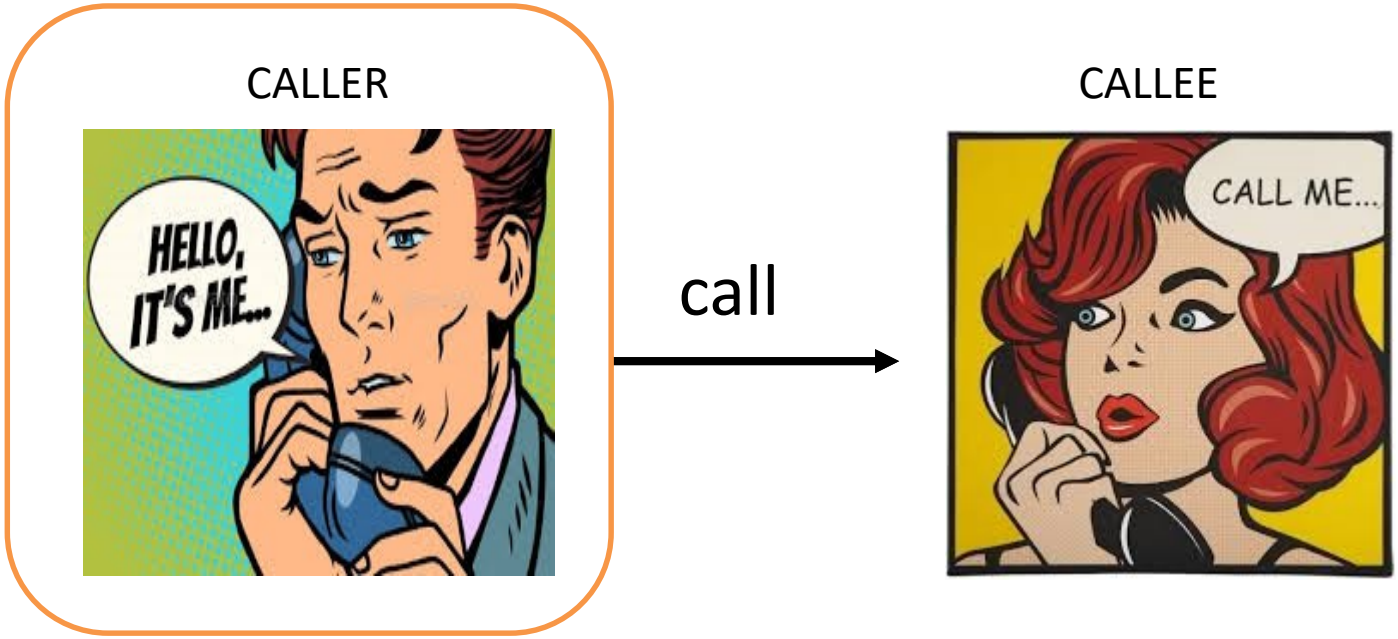
Caller Instructions

```
set_arg k, x
call p
retrieve x
```

(move x to parameter slot k)

(transfer control to p)

(*Put return value in X*)



3AC: Caller code example

```
int a = bad_max(b, 7);
```

set_arg 1 b
set_arg 2 7
call bad_max
retrieve a

`set_arg k, x`	*(move x to parameter slot k)*
`call p`	*(transfer control to p)*
`retrieve x`	(*Put return value in X*)

3AC Function Calls

```
int bad_max(int a1, int a2)
    return a2 + 1


Int a = bad_max(b, 7)
```

Enter bad_max
a1 = get_arg 1
a2 = get_arg 2
leave bad_max
return a2+1

Callee Instructions

`enter p`	*(enter function scope)*
`x = get_arg k`	*(retrieve argument k)*
`leave p`	(Leave function scope)
`return x`	*(Set return value)*

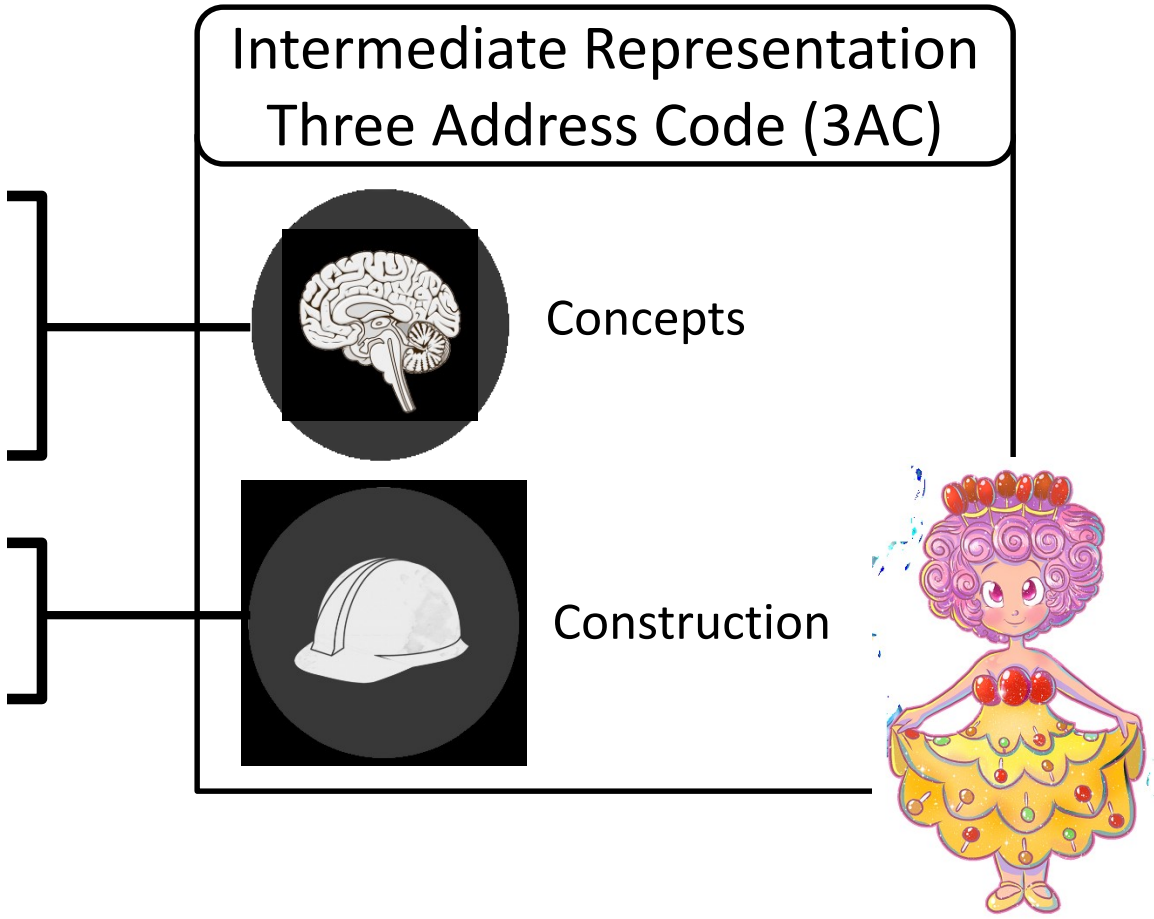
Summary

Call/Return semantics largely encoded in 3AC instructions

This Lecture

- 3AC: Definition
- 3AC: Intuition
- 3AC: Examples

- From AST to 3AC



From AST to 3AC

- Simply another postorder traversal of the tree
  - Have to be a little bit careful about order
    - Always do the RHS of assignment before the LHS, for example

Next Lecture

- Look at Control-Flow Graphs (The *other* CFGs)
  - A hybrid IR
    - Combines some structural overlay on a linear representation