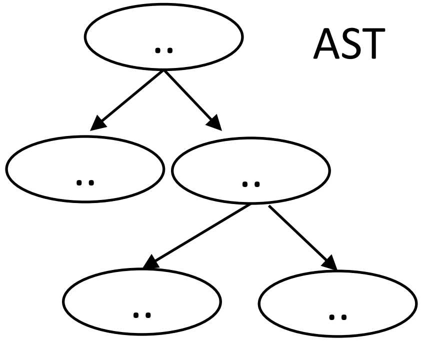
CSE110A: Compilers

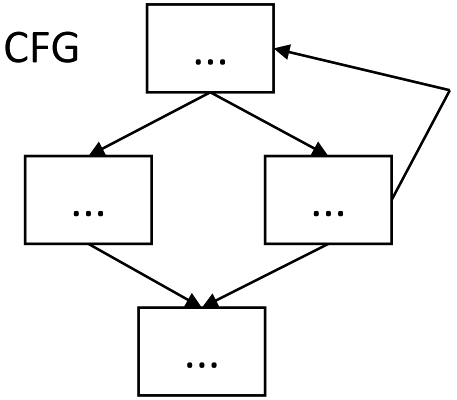
AST



Topics:

- *Module 3: Intermediate representations*
  - *Finishing up type checking*
  - *Linear Irs : 18*

CFG



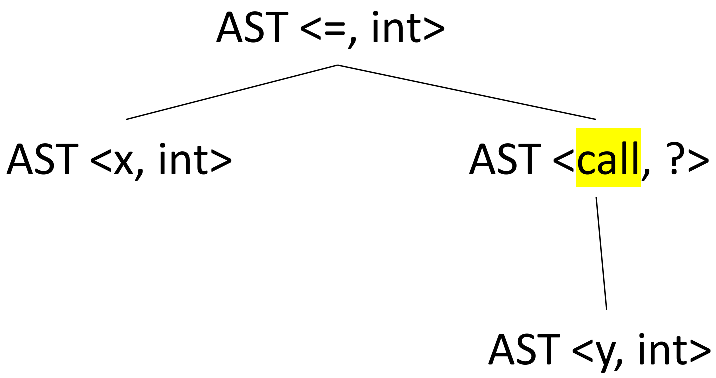
3 address code

```
store i32 0, ptr %2
%3 = load i32, ptr %1
%4 = add nsw i32 %3, 1,
store i32 %4, ptr %1
%5 = load i32, ptr %2
```

Type Systems

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```



requires a function specification,
using in the .h file:

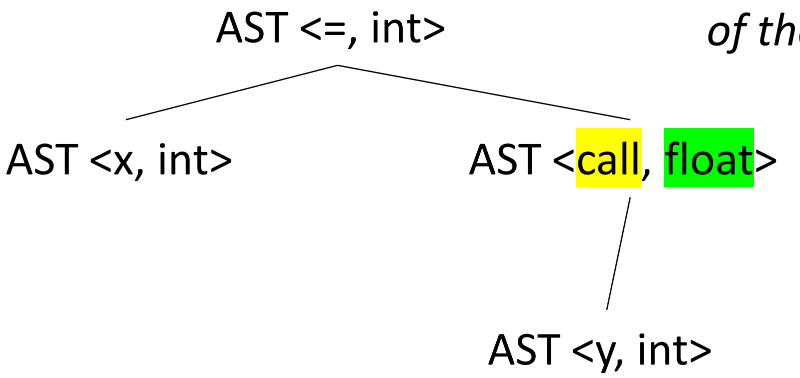
```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```

type of the AST node becomes the return type of the function



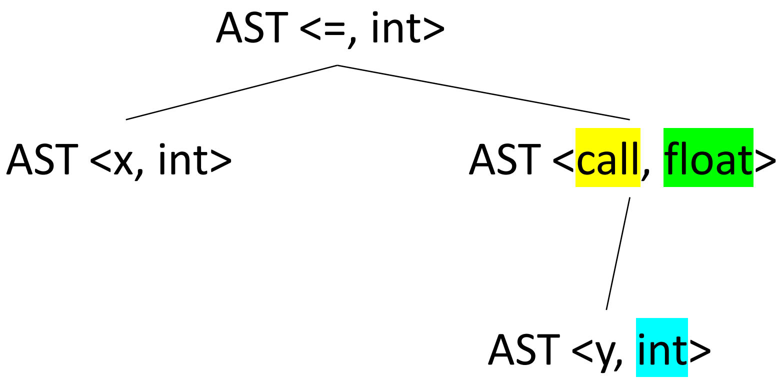
requires a function specification,
using in the .h file:

```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```



type inference must make sure arguments match types

requires a function specification,
using in the .h file:

```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```

AST <=, int>

AST <x, int>

AST <call, float>

AST <int2float, float>

AST <y, int>

type inference must make sure arguments match types

requires a function specification, using in the .h file:

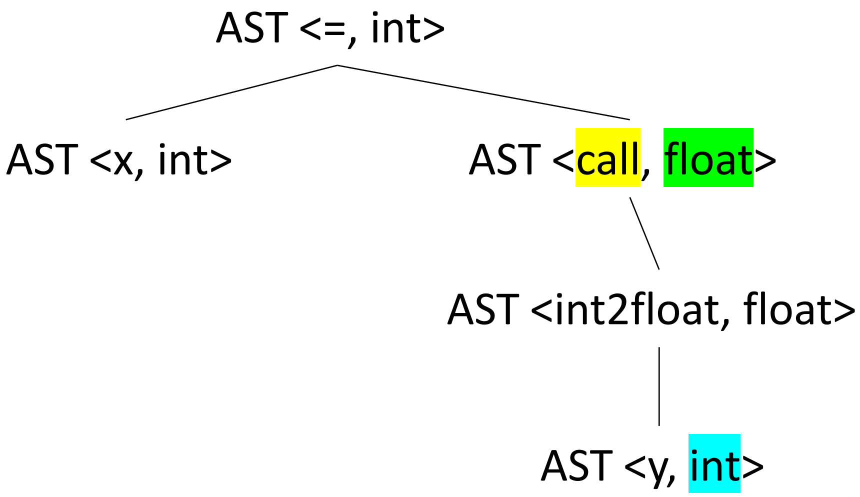
```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```

How would type inference finish this?



requires a function specification,
using in the .h file:

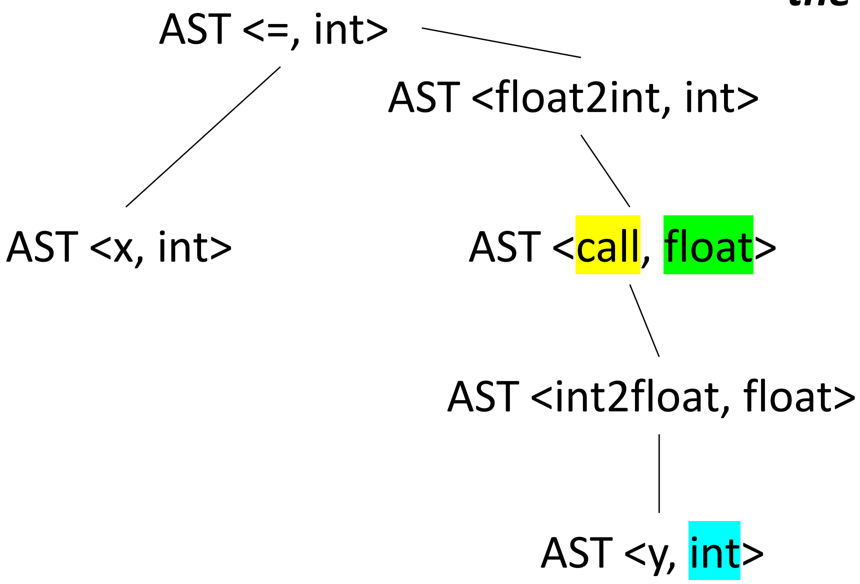
```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```

How would type inference finish this?
remember that assignment converts to the lhs type



requires a function specification,
using in the .h file:

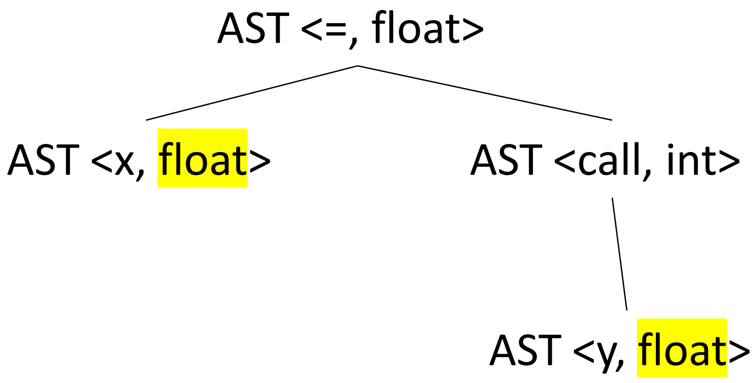
```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

What about floats to ints?

```
int int_sqrt(int input);

float x;
float y;
x = int_sqrt(y)
```

Does this compile?

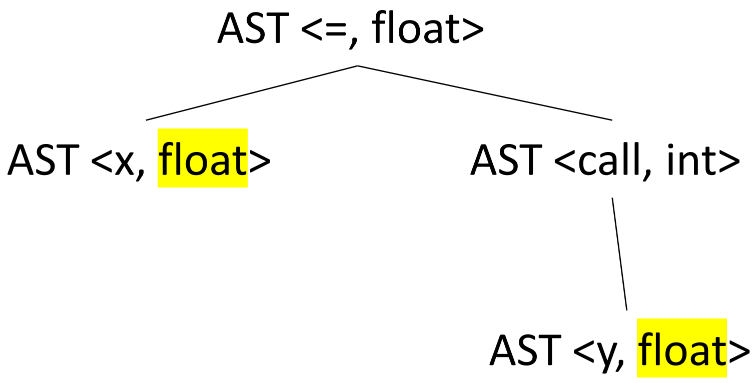


What about floats to ints?

```
int int_sqrt(int input);

float x;
float y;
x = int_sqrt(y)
```

Does this compile? Yes!

In this case the compiler will convert floats to an int.
Is that the right choice? ...

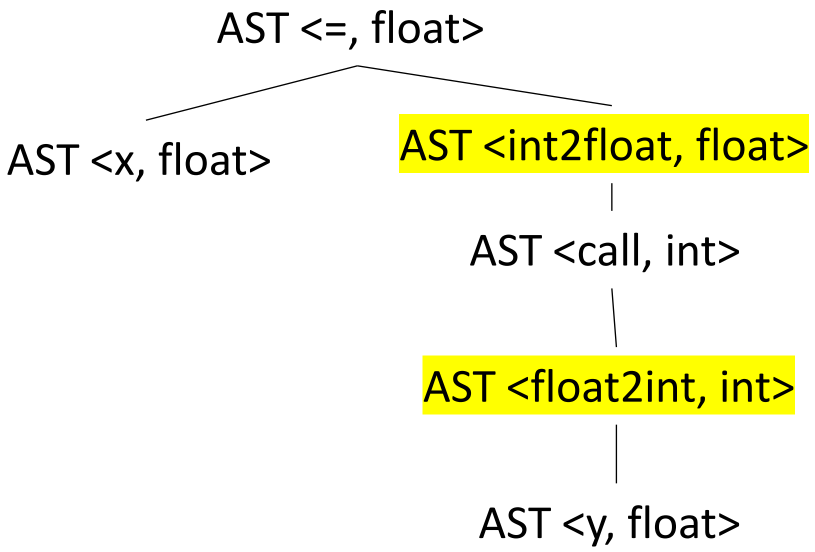


What about floats to ints?

```
int int_sqrt(int input);

float x;
float y;
x = int_sqrt(y)
```

Does this compile? Yes!

In this case the compiler will convert floats to an int.
Is that the right choice? ...



Discussion

- Many languages (and styles) state that the programmer extends the type system through functions

- Other languages allow operator overloading
  - Controversial design pattern
  - But it can be really nice (e.g. it is used extensively in LLVM internals)

```
class Complex {
  private:
   float real;
   float imag;
  public:
   // Constructor to initialize real and imag to 0
   Complex() : real(0), imag(0) {}

   // Overload the + operator
   Complex operator + (const Complex& obj) {
     Complex temp;
     temp.real = real + obj.real;
     temp.imag = imag + obj.imag;
     return temp;
   }
```

Table for ***plus*** binary ops

left child	right child	result
int	int	int
int	float	float
float	int	float
float	float	float
Complex	Complex	Complex

code from: https://www.programiz.com/cpp-programming/operator-overloading

```
class Complex {
  private:
   float real;
   float imag;
  public:
   // Constructor to initialize real and imag to 0
   Complex() : real(0), imag(0) {}

   // Overload the + operator
   Complex operator + (const Complex& obj) {
     Complex temp;
     temp.real = real + obj.real;
     temp.imag = imag + obj.imag;
     return temp;
   }


      Complex operator + (const float& i) {
        Complex temp;
        temp.real = real + i;
        temp.imag = imag;
        return temp;
      }
```

Table for ***plus*** binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`
`Complex`	`Complex`	`Complex`

code from: https://www.programiz.com/cpp-programming/operator-overloading

```
class Complex {
  private:
   float real;
   float imag;
  public:
   // Constructor to initialize real and imag to 0
   Complex() : real(0), imag(0) {}

   // Overload the + operator
   Complex operator + (const Complex& obj) {
     Complex temp;
     temp.real = real + obj.real;
     temp.imag = imag + obj.imag;
     return temp;
   }

      Complex operator + (const float& i) {
         Complex temp;
         temp.real = real + i;
         temp.imag = imag;
         return temp;
       }
```

Table for ***plus*** binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`
`Complex`	`Complex`	`Complex`
`Complex`	`float`	`Complex`

We can add extra rows

code from: https://www.programiz.com/cpp-programming/operator-overloading

Type systems finished

- Defined what a type system is and discussed various different design decisions
  - static vs. dynamic, choice of primitive types, size of primitive types
- Implemented type inference parameterized by type conversion tables on an AST.
  - identified common conversions (int to float) and when the opposite can happen
- Discussed how programmers can extend the type system
  - function calls
  - operator overloading

Intermediate Representations

Our next IR: 3 address code or linear IR

- We will specify our own that we will use in this class
  - Will be used in the next two homeworks
- Similar to assembly
  - Untyped
  - Specialized operations for each type
- Similar to typical IRs (e.g. LLVM)
  - Unlimited virtual registers

Class-IR

Inputs/outputs (IO): 32-bit typed inputs

e.g.: `int x, int y, float z`

Program Variables (Variables): 32-bit untyped virtual register

given as `vrX` where `X` is an integer:

e.g. `vr0, vr1, vr2, vr3` …

we will assume input/output names are disjoint from virtual register names

Class-IR

binary operators:

```
dst = operation(op0, op1);
```

```
operations can be one of:
[add, sub, mult, div, eq, lt]
```

```
each operation is followed by an i or f, which
specifies how the bits in the registers are
interpreted
```

Class-IR

binary operators:

```
dst = operation(op0, op1);
```

```
operations can be one of:
[add, sub, mult, div, eq, lt]
```

all of dst, op0, and op1 must be untyped virtual registers.

Class-IR

binary operators:

```
dst = operation(op0, op1);

Examples:

vr0 = addi(vr1, vr2);
vr3 = subf(vr4, vr5);

x = multf(vr0, vr1); not allowed!
vr0 = addi(vr1, 1);  not allowed!
```

We'll talk about how to do this using other instructions

Class-IR

Control flow

`branch(label);`

- branches unconditionally to the label

`bne(op0, op1, label)`

- if op0 is not equal to op1 then branch to label
- operands must be virtual registers!

`beq(op0, op1, label)`

- Same as bne except it is for equal

Class-IR

Assignment

```
vr0 = vr1
```

```
one virtual register can be assigned to another
```

Class-IR

Assignment

```
vr0 = vr1

one virtual register can be assigned to another

Examples:
vr0 = 1; not allowed
vr1 = x; not allowed
```

Class-IR

`unary get untyped register`

```
dst = operation(op0);

operations are: [int2vr, float2vr]

Example:
```

Given IO: int x

```
vr1 = int2vr(x);
vr2 = float2vr(2.0);
```

Class-IR

unary get typed data

```
dst = operation(op0);

operations are: [vr2int, vr2float]

Example:
```

Given IO: int x and float y

```
x = vr2int(vr1);
y = vr2float(vr3);
```

Class-IR

unary conversion operators:

```
dst = operation(op0);
```

```
operations can be one of:
[vr_int2float, vr_float2int]
```

`converts the bits in a virtual register from one type to another.` *`op0 and dst must be a virtual register!`*

Class-IR

unary conversion operators:

```
dst = operation(op0);

Examples:

vr0 = vr_int2float(vr1);
vr2 = vr_float2int(1.0); not allowed!
```

Example

`adding the values 1 - 9 to an input/output variable: int x`

Example

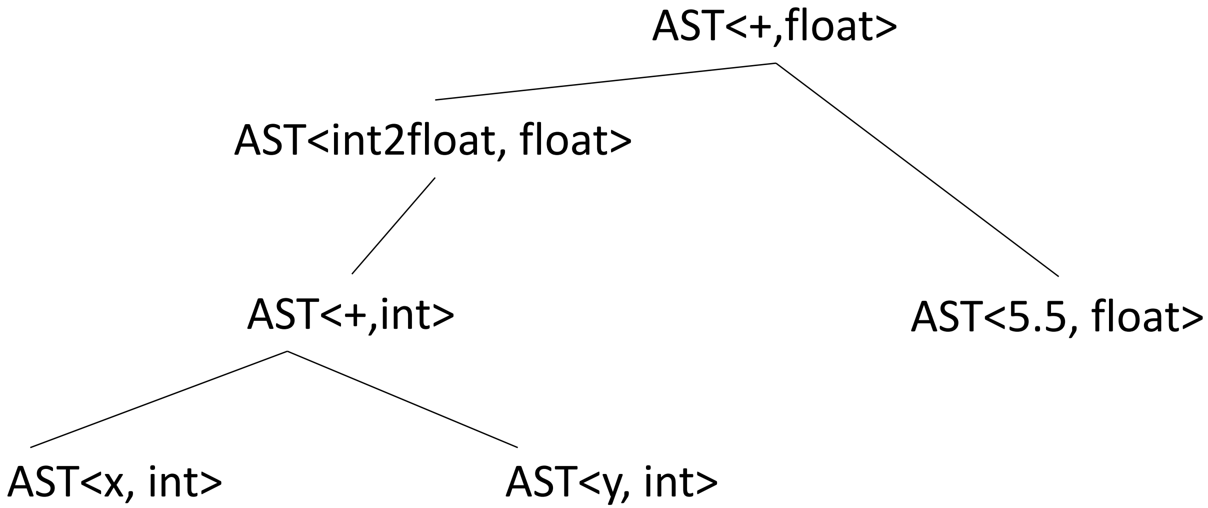
```
adding the values 1 - 9 to an input/output variable: int x

  vr0 = int2vr(1);
  vr1 = int2vr(1);
  vr2 = int2vr(10);
loop_start:
  vr3 = lti(vr0, vr2);
  bne(vr3, vr1, end_label);
  vr4 = int2vr(x);
  vr5 = addi(vr4,vr0);
  x = vr2int(vr5);
  vr0 = addi(vr0, vr1);
  branch(loop_start);
end_label:
```

Converting AST into Class-IR

Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```

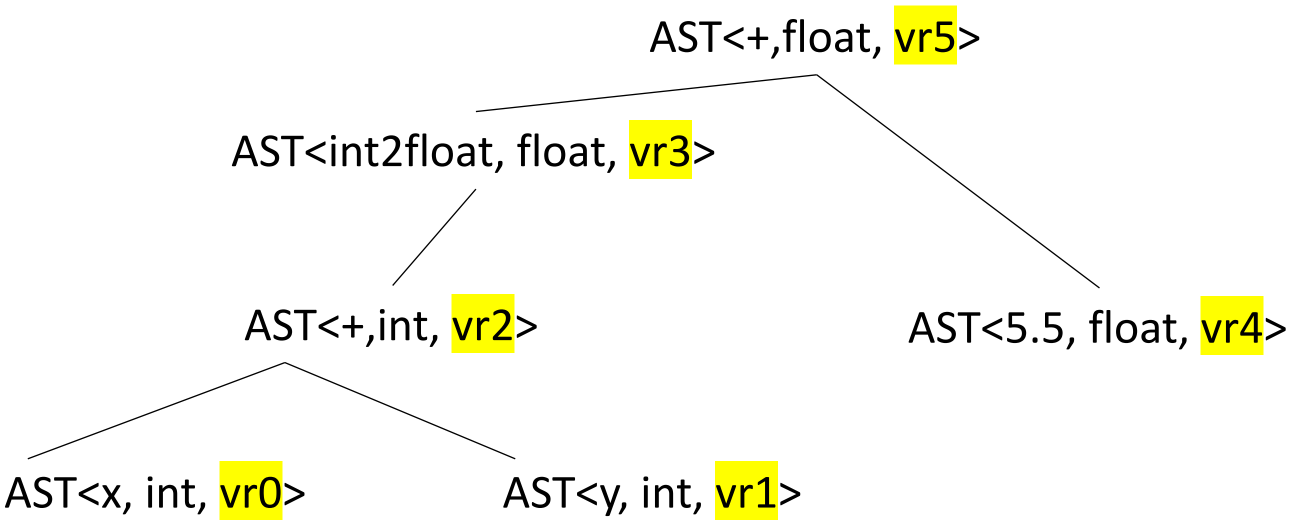
After type inference



Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```

After type inference



We will start by adding a new member to each AST node:

A virtual register

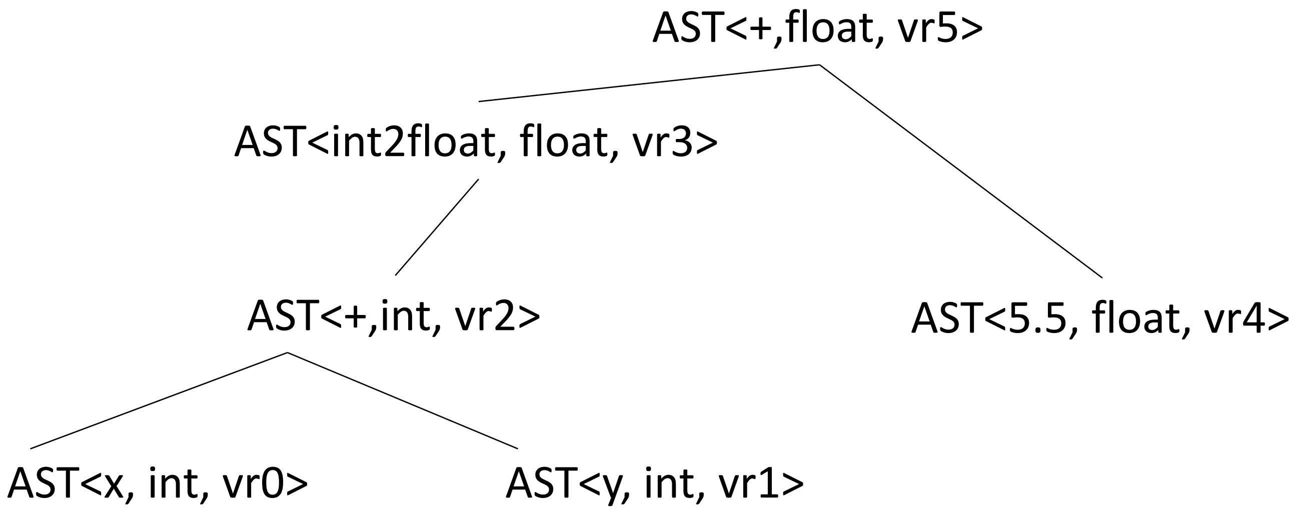
Each node needs a distinct virtual register

Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```

After type inference

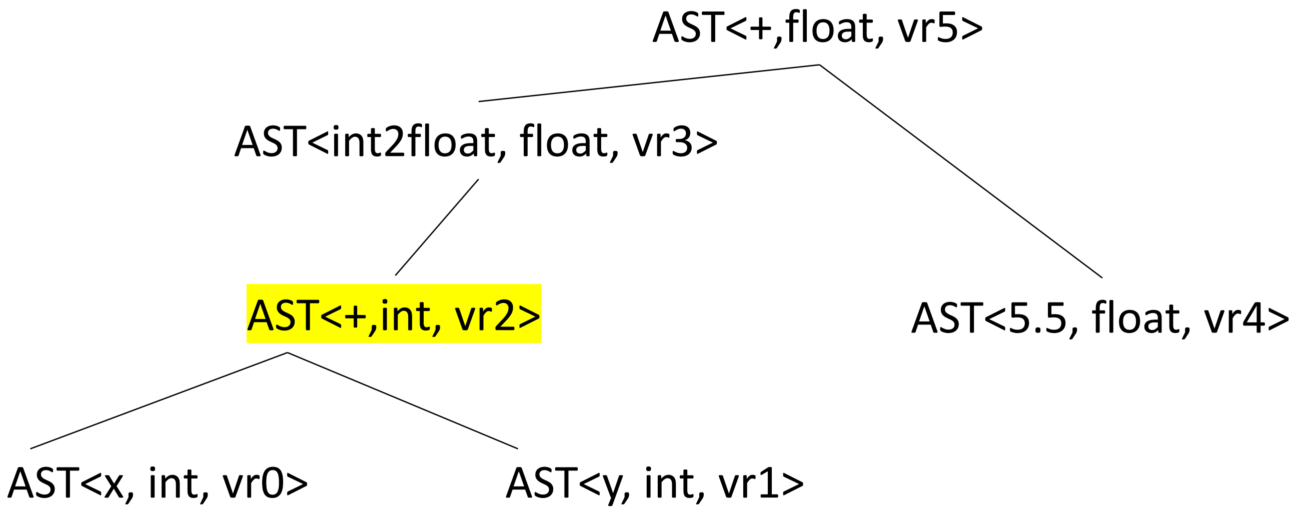
Next each AST node needs to know how to print a 3 address instruction



Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```

After type inference



Next each AST node needs to know how to print a 3 address instruction

Let's look at add

```
class ASTPlusNode(ASTBinOpNode):
  def __init__(self, l_child, r_child):
    super().__init__(l_child,r_child)

  # return a string of the three address instruction
  # that this node encodes
  def three_addr_code(self):
    ??
```

```
return "%s = %s(%s,%s);" %
          (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

```
class ASTPlusNode(ASTBinOpNode):
    def __init__(self, l_child, r_child):
        super().__init__(l_child,r_child)

    # return a string of the three address instruction
    # that this node encodes
    def three_addr_code(self):
        ??
```

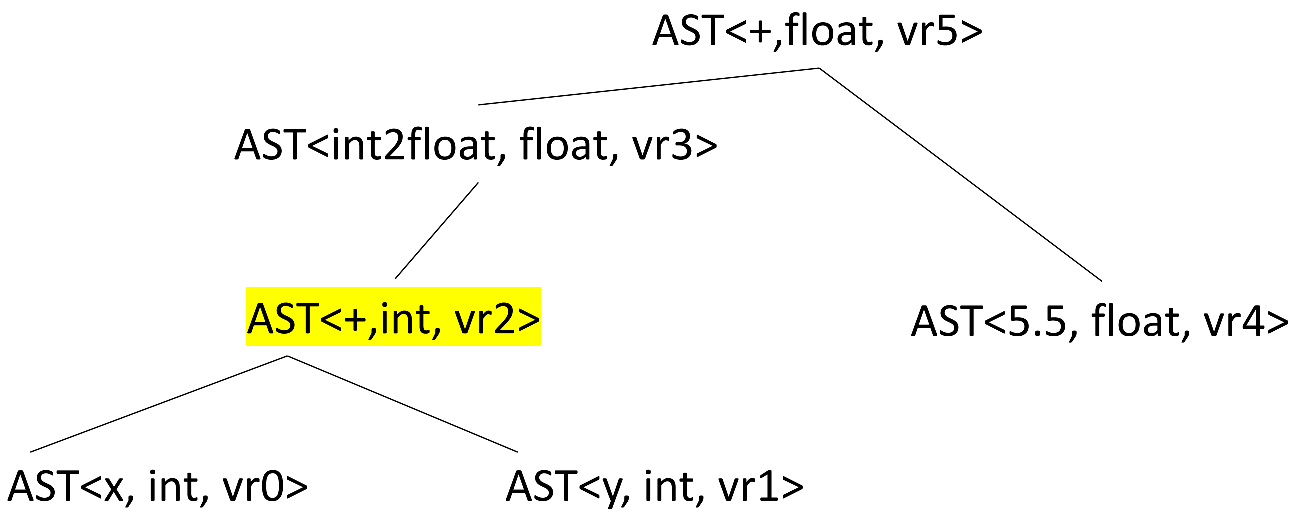
```
return "%s = %s(%s,%s);" %
        (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

What is this one?

```python
def get_op(self):
    if self.node_type is Types.INT:
        return "addi"
    else:
        return "addf"
```

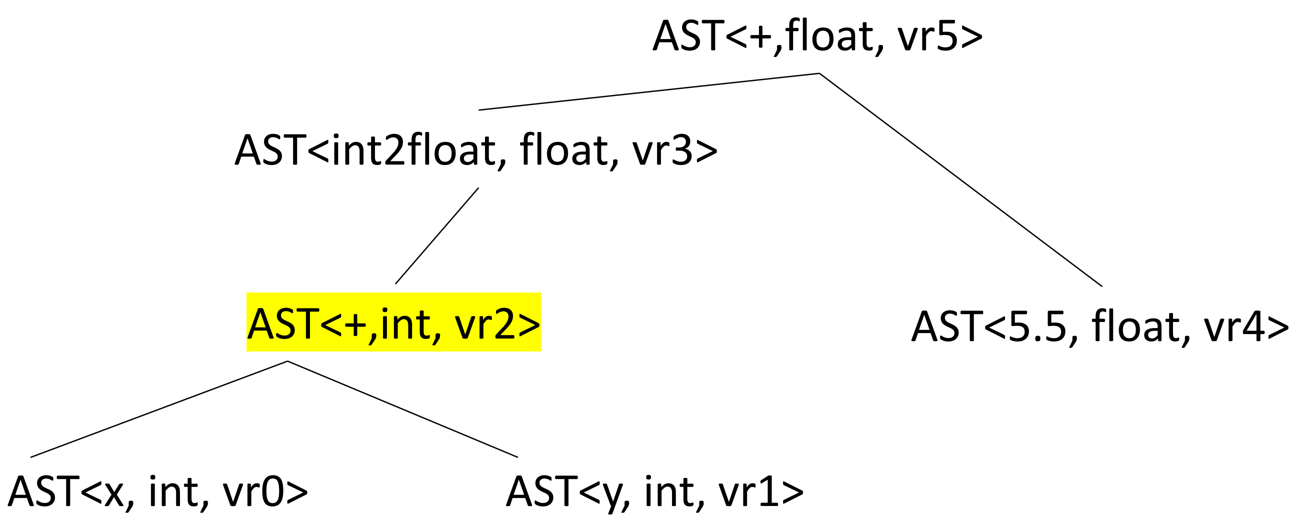
```python
return "%s = %s(%s,%s);" %
        (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

What is this one?




```python
def get_op(self):
    if self.node_type is Types.INT:
        return "addi"
    else:
        return "addf"
```

```python
return "%s = %s(%s,%s);" %
        (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

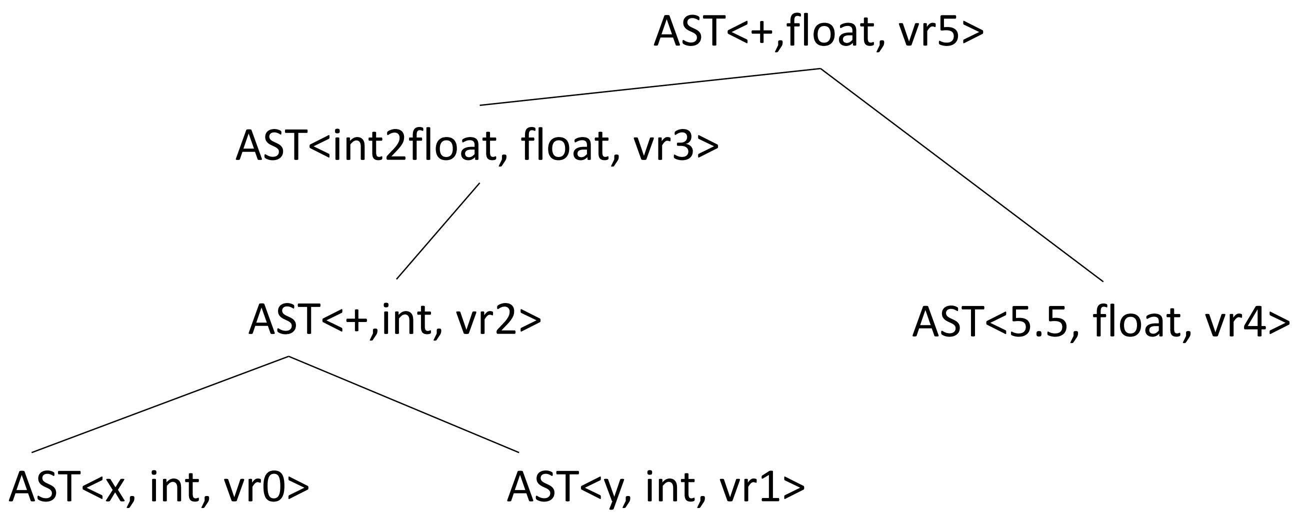



```
def get_op(self):
    if self.node_type is Types.INT:
        return "addi"
    else:
        return "addf"
```

```
return "%s = %s(%s,%s);" %
        (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

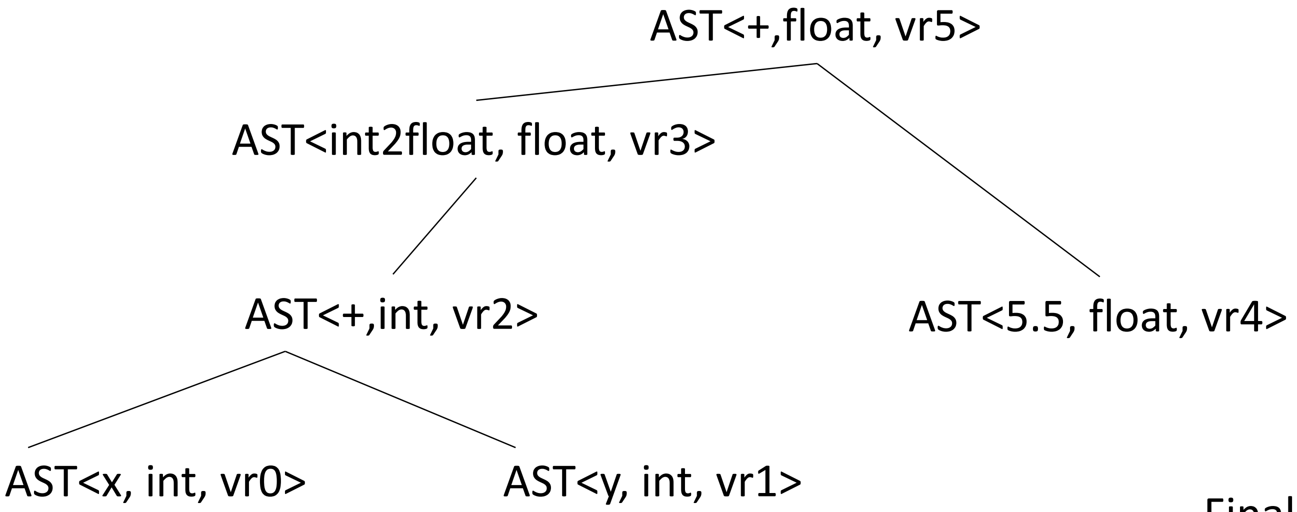
```
vr2 = addi(vr0,vr1);
```

```
int x;
int y;
float w;
w = x + y + 5.5
```




```
vr0 = int2vr(x);
vr1 = int2vr(y);
vr4 = float2vr(5.5);
vr2 = addi(vr0,vr1);
vr3 = vr_int2float(vr2);
vr5 = addf(vr3,vr4);
```

```
int x;
int y;
float w;
w = x + y + 5.5
```

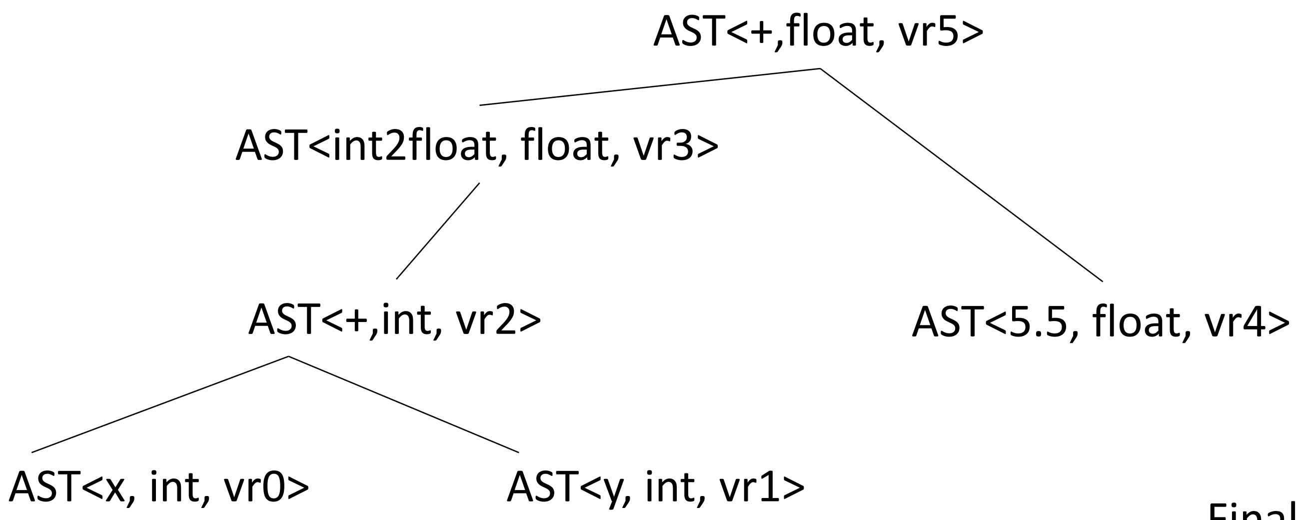


Final program

```
vr0 = int2vr(x);
vr1 = int2vr(y);
vr2 = addi(vr0,vr1);
vr3 = vr_int2float(vr2);
vr4 = float2vr(5.5);
vr5 = addf(vr3,vr4);
```

We can create a 3 address program doing a post-order traversal

```
int x;
int y;
float w;
w = x + y + 5.5
```



Final program

```
vr0 = int2vr(x);
vr1 = int2vr(y);
vr2 = addi(vr0,vr1);
vr3 = vr_int2float(vr2);
vr4 = float2vr(5.5);
vr5 = addf(vr3,vr4);
```

We can create a 3 address program doing a post-order traversal

Is this the only ordering?

Backing-up to an even higher level

We now know how to:

- parse an expression: `parse_expr`
- create an AST during parsing
- do type inference on an AST
- convert a type-safe AST into 3 address code

Backing up to an even higher level

- We can now define what our parser will return: A list of 3 address code
- We can get 3 address code from parsing expressions, now we just need to get it from statements

From our grammar

```
statement := declaration_statement
          |  assignment_statement
          |  if_else_statement
          |  block_statement
          |  for_loop_statement
```

Our top-down parser should have a function called `parse_statement`

This should return a list of 3 address code instructions that encode the statement

From our grammar

```
statement := declaration_statement
           |  assignment_statement
           |  if_else_statement
           |  block_statement
           |  for_loop_statement
```

Our top down parser should have a function called `parse_statement`

This should return a list of 3 address code instructions that encode the statement

```
int x;
int y;
float w;
w = x + y + 5.5
```

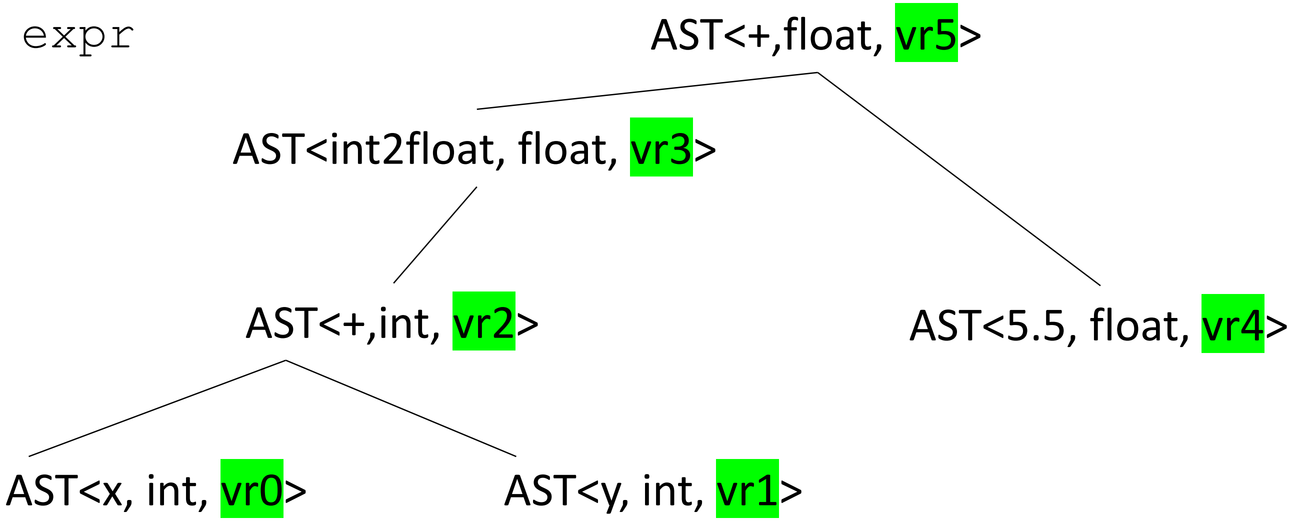
```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match.value
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % ?
    return program + [new_inst]
}
```

```
int x;
int y;
float w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr
```

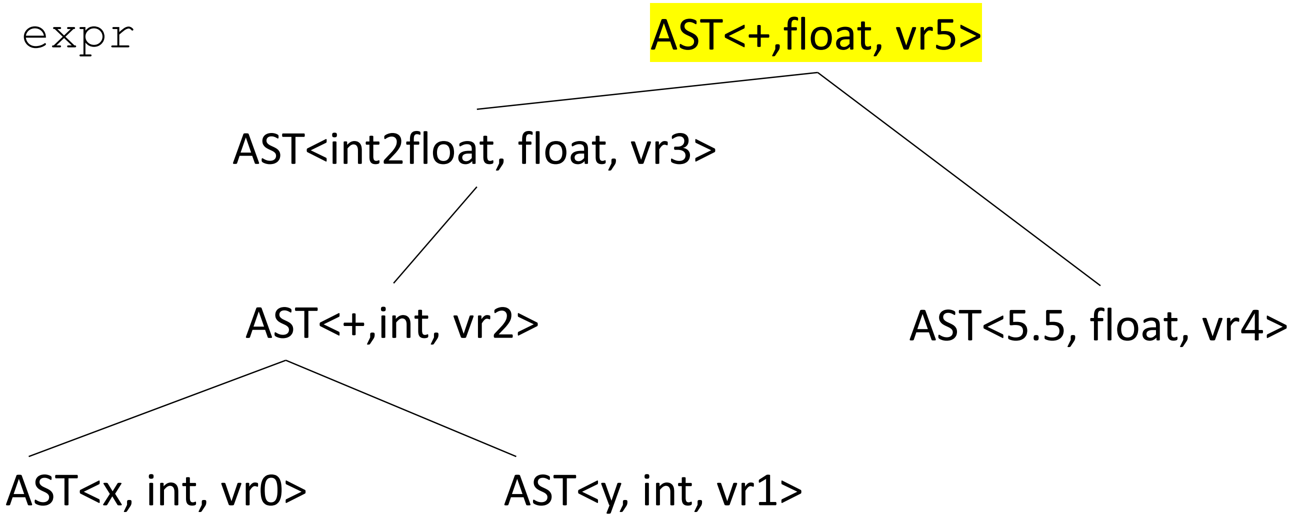
```
{
   id_name = to_match.value
   eat(“ID”);
   eat(“ASSIGN”);
   ast = parse_expr()
   type_inference(ast)
   assign_registers(ast)
   program = ast.linearize()
   new_inst = “%s = %s” % ?
   return program + [new_inst]
}
```



```
int x;
int y;
float w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr
```

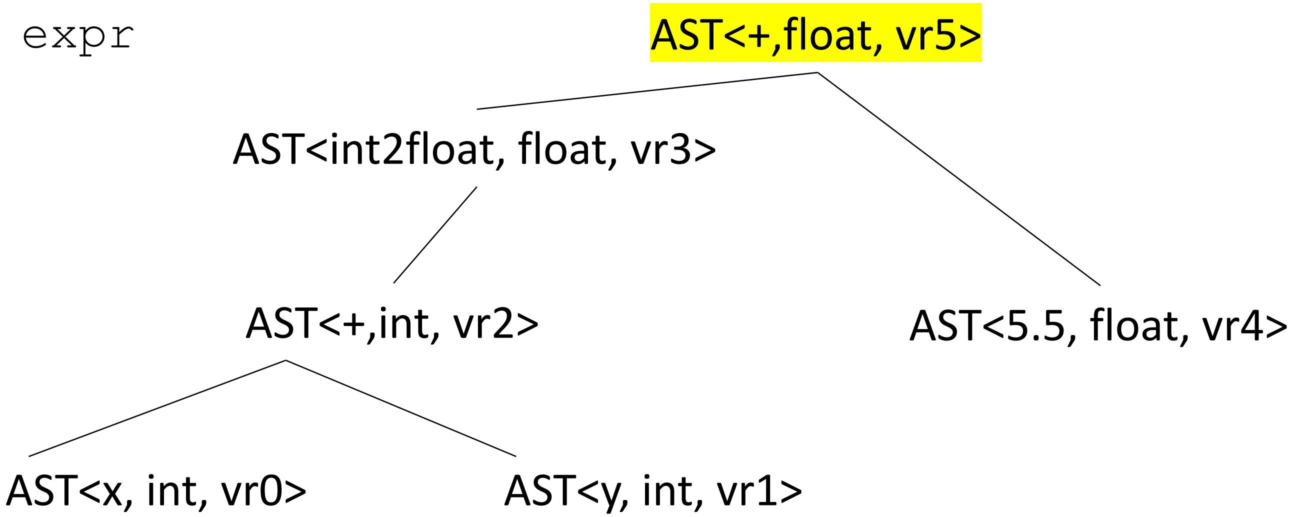
```
{
    id_name = to_match.value
    eat(“ID”);
    eat(“ASSIGN”);
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = “%s = %s” % ?
    return program + [new_inst]
}
```



```
int x;
int y;
float w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match.value
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
}
```



```
int x;
int y;
float w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match.value
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
}
```

program

```
vr0 = int2vr(x);
vr1 = int2vr(y);
vr2 = addi(vr0,vr1);
vr3 = vr_int2float(vr2);
vr4 = float2vr(5.5);
vr5 = addf(vr3,vr4);
```

new inst

```
w = vr5
```

```
int x;
int y;
float w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr
```

```
{
   id_name = to_match.value
   eat("ID");
   eat("ASSIGN");
   ast = parse_expr()
   type_inference(ast)
   assign_registers(ast)
   program = ast.linearize()
   new_inst = "%s = %s" % (id_name, ast.vr)
   return program + [new_inst]
}
```

What are we missing here?

1. If the type of ID doesn't match the type of the ast, then the ast needs to be converted.

2. ID should be checked if it is an input/output variable. which means it will need to be handled differently.

3. You need to check the ID in the symbol table

it can get a little messy

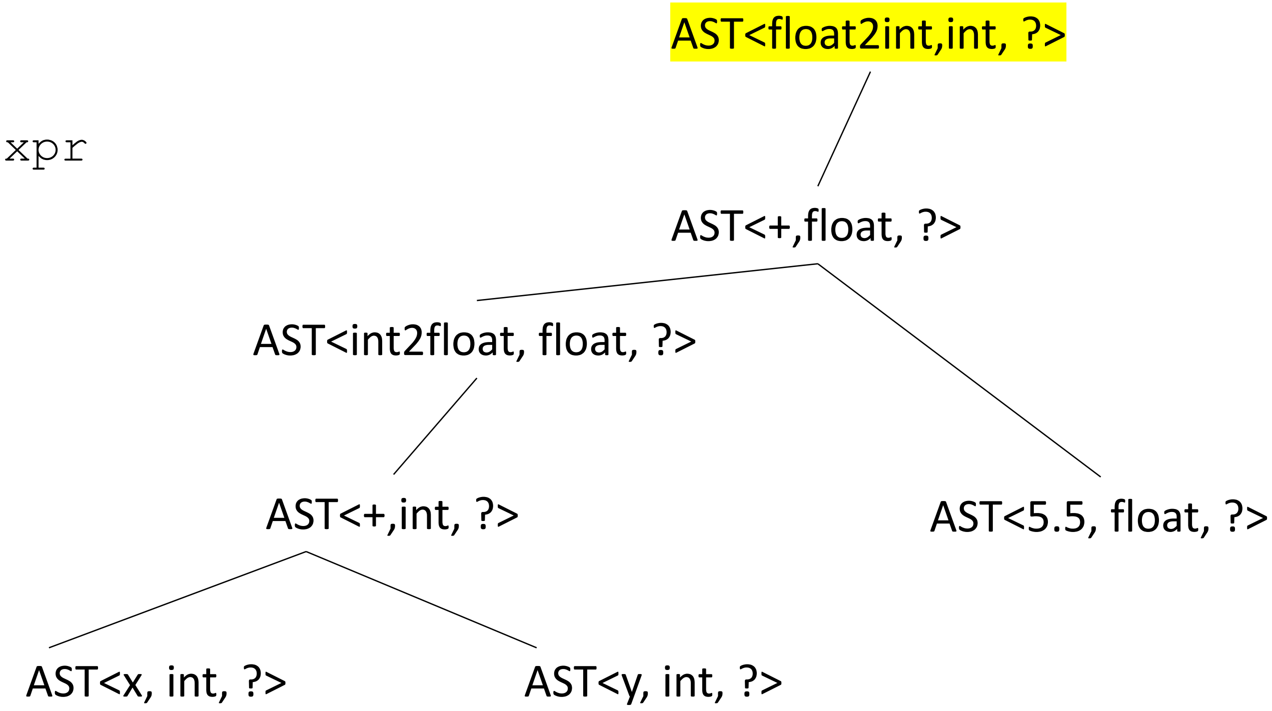
```
int x;
int y;
int w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr

{
    id_name = to_match.value
    id_data_type = # get ID data type
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                  ast.node_type == FLOAT:
       ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
}
```

one possible case

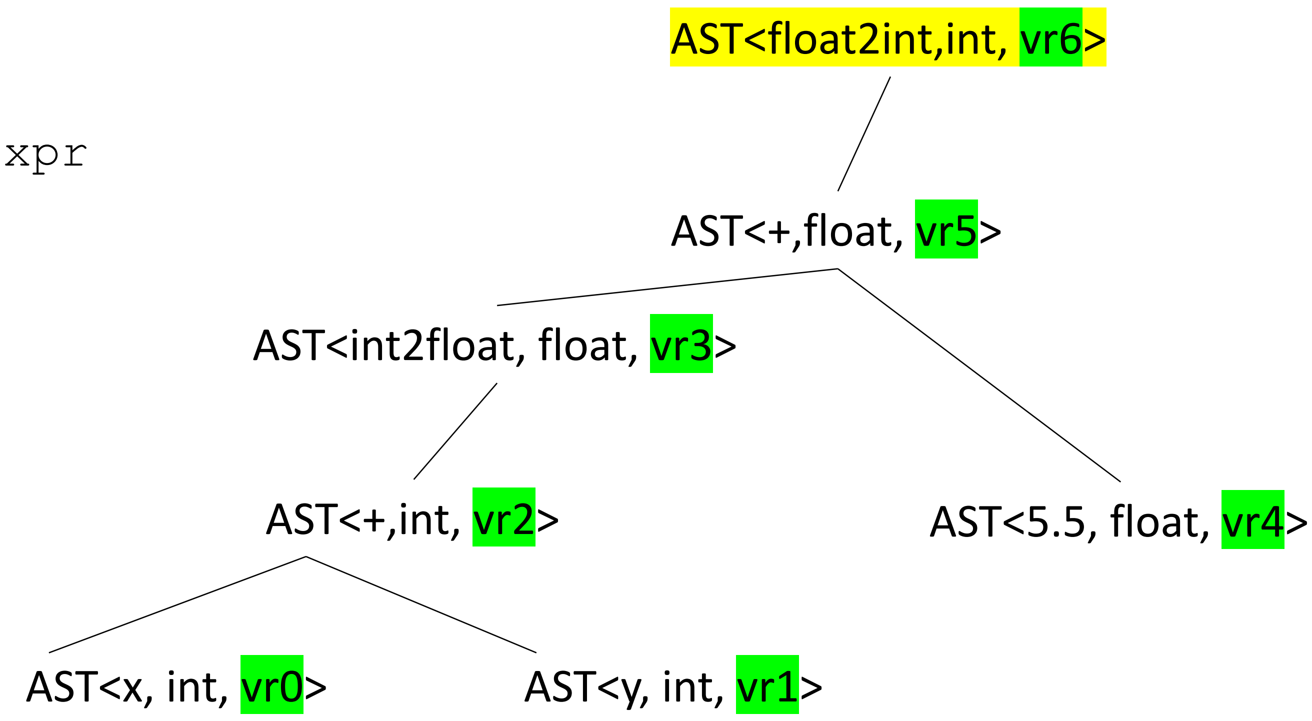
```
int x;
int y;
int w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                   ast.node_type == FLOAT:
       ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
 }
```



```
int x;
int y;
int w;
w = x + y + 5.5
```

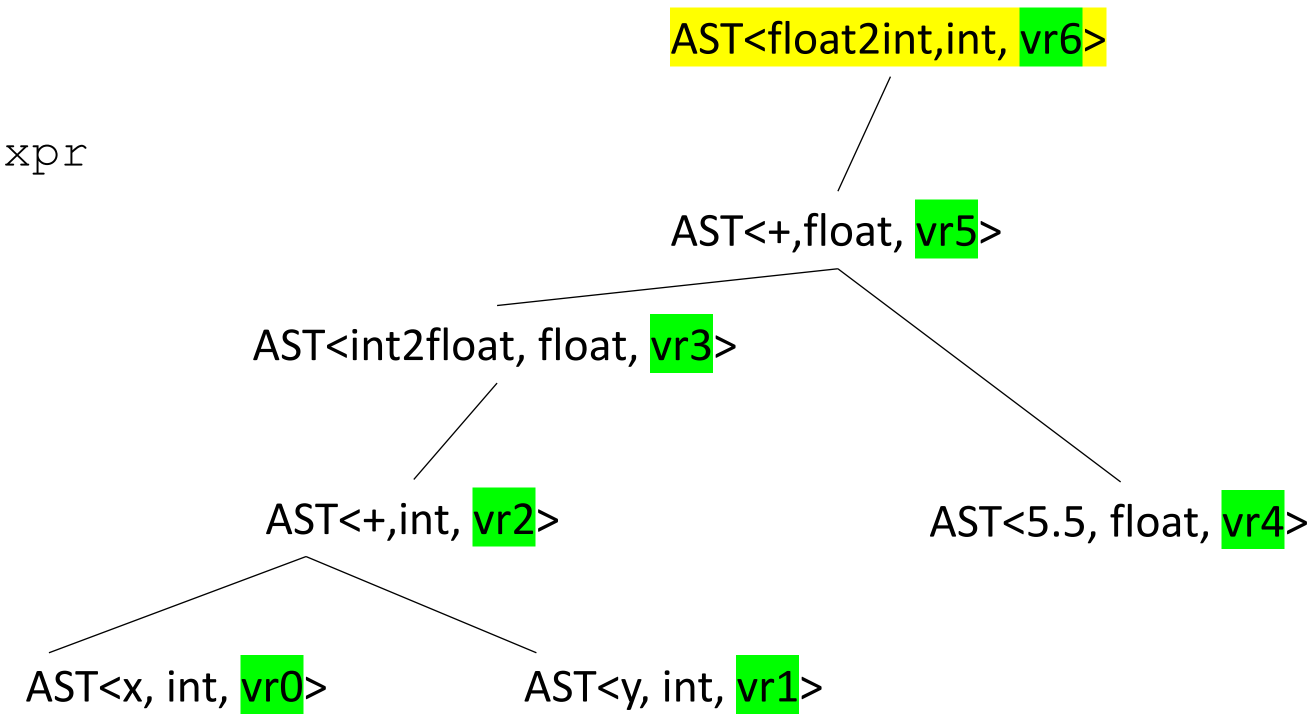
```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                    ast.node_type == FLOAT:
        ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
 }
```



```
(IO: int w)
int x;
int y;
w = x + y + 5.5
```

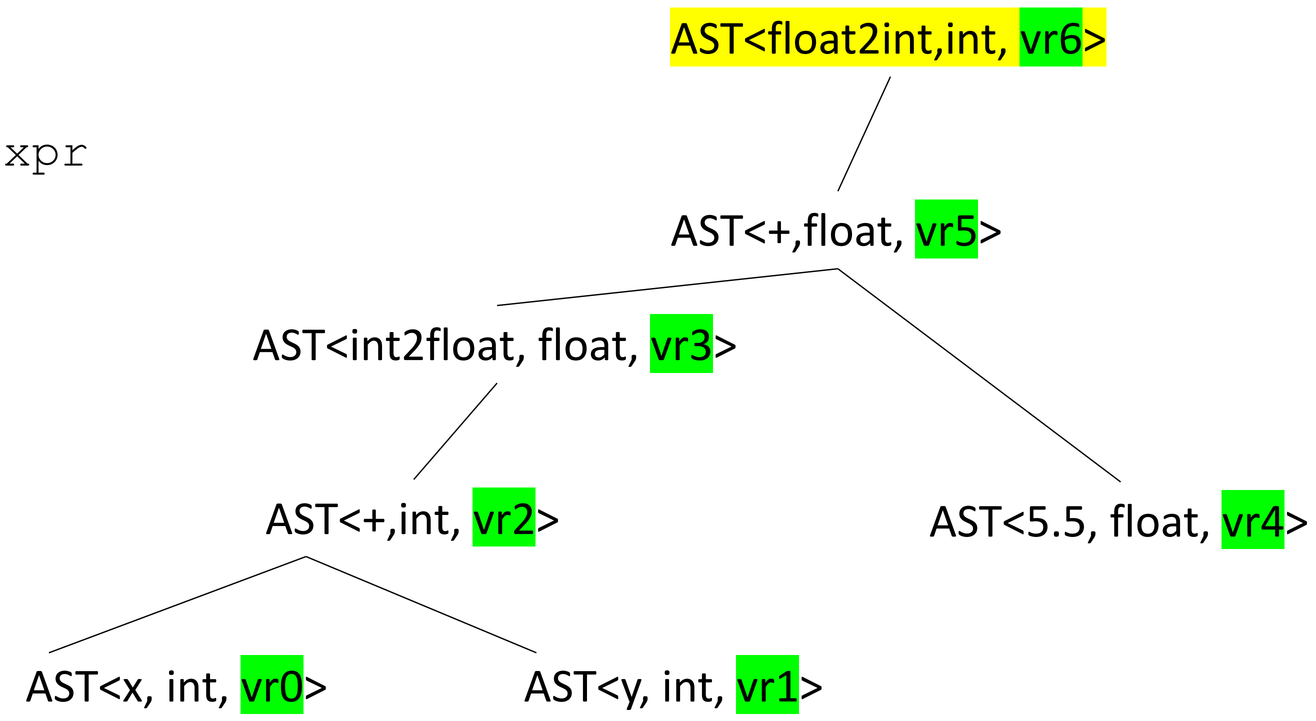
How would we deal with w as an IO variable?

```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                ast.node_type == FLOAT:
        ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
 }
```



```
(IO: int w)
int x;
int y;
w = x + y + 5.5
```

How would we deal with w as an IO variable?




```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                  ast.node_type == FLOAT:
          ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = vr2int(%s)" % (id_name, ast.vr)
    return program + [new_inst]
 }
```

Only if it is an IO variable!

It gets a little messy

Let’s do another one

```
statement := declaration_statement
           |  assignment_statement
           |  if_else_statement
           |  block_statement
           |  for_loop_statement
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    eat("IF");
    eat("LPAR");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

```
if (program0) {
   program1
}
else {
   program2
}
```

We need to convert this to 3 address code

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    eat("IF");
    eat("LPAR");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

```
if (program0) {
   program1
}
else {
   program2
}
```

We need to convert this to 3 address code

```
program0
program1
program2
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    eat("IF");
    eat("LPAR");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

```
if (program0) {
  program1
}
else {
  program2
}
```

We need to convert this to 3 address code

```
  program0;
  vrX = int2vr(0)
  beq(expr_ast.vr, vrX, else_label);
  program1
  branch(end_label);
else_label:
  program2
end_label:
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    ...
    # get resources
    end_label  = mk_new_label()
    else_label = mk_new_label()
    vrX        = mk_new_vr()

    # make instructions
    ins0 = "%s = int2vr(0)" % vrX
    ins1 = "beq(%s, %s, %s);" %
           (expr_ast.vr, vrX, else_label)
    ins2 = "branch(%s)" % end_label

    # concatenate all programs
    return program0 + [ins0, ins1] + program1
           + [ins2, label_code(else_label)]
           + program2 + [label_code(end_label)]
}
```

```
if (program0) {
   program1
}
else {
   program2
}
```

We need to convert this to 3 address code

```
  program0;
  vrX = int2vr(0)
  beq(expr_ast.vr, vrX, else_label);
  program1
  branch(end_label);
else_label:
  program2
end_label:
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    ...
    # get resources
    end_label  = mk_new_label()
    else_label = mk_new_label()
    vrX        = mk_new_vr()

    # make instructions
    ins0 = "%s = int2vr(0)" % vrX
    ins1 = "beq(%s, %s, %s);" %
           (expr_ast.vr, vrX, else_label)
    ins2 = "branch(%s)" % end_label

    # concatenate all programs
    return program0 + [ins0, ins1] + program1
           + [ins2, label_code(else_label)]
           + program2 + [label_code(end_label)]
}
```

```python
class VRAllocator():
    def __init__(self):
        self.count = 0


    def get_new_register(self):
        vr = "vr" + str(self.count)
        self.count += 1
        return vr
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    ...
    # get resources
    end_label   = mk_new_label()
    else_label  = mk_new_label()
    vrX         = mk_new_vr()

    # make instructions
    ins0 = "%s = int2vr(0)" % vrX
    ins1 = "beq(%s, %s, %s);" %
           (expr_ast.vr, vrX, else_label)
    ins2 = "branch(%s)" % end_label

    # concatenate all programs
    return program0 + [ins0, ins1] + program1
           + [ins2, label_code(else_label)]
           + program2 + [label_code(end_label)]
}
```

```
class LabelAllocator():
    def __init__(self):
        self.count = 0


    def get_new_register(self):
        lb = "label" + str(self.count)
        self.count += 1
        return lb
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
   ...
   # get resources
   end_label  = mk_new_label()
   else_label = mk_new_label()
   vrX        = mk_new_vr()

   # make instructions
   ins0 = "%s = int2vr(0)" % vrX
   ins1 = "beq(%s, %s, %s);" %
          (expr_ast.vr, vrX, else_label)
   ins2 = "branch(%s)" % end_label

   # concatenate all programs
   return program0 + [ins0, ins1] + program1
          + [ins2, label_code(else_label)]
          + program2 + [label_code(end_label)]
}
```

```
  program0;
  vrX = int2vr(0)
  beq(expr_ast.vr, vrX, else_label);
  program1
  branch(end_label);
else_label:
  program2
end_label:
```

Need a :

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
   ...
   # get resources
   end_label  = mk_new_label()
   else_label = mk_new_label()
   vrX        = mk_new_vr()

   # make instructions
   ins0 = "%s = int2vr(0)" % vrX
   ins1 = "beq(%s, %s, %s);" %
          (expr_ast.vr, vrX, else_label)
   ins2 = "branch(%s)" % end_label

   # concatenate all programs
   return program0 + [ins0, ins1] + program1
          + [ins2, label_code(else_label)]
          + program2 + [label_code(end_label)]
}
```

```
def label_code(l): return l + ":"
```

```
statement := declaration_statement
          |  assignment_statement
          |  if_else_statement
          |  block_statement
          |  for_loop_statement
```

We did these two

You do these two for your homework

Draw out for loops just like how we did with the if statements!

Compiler pragmatics

- New terminology I learned recently:
  - Implementation details

- We need to talk about different ID types (IO, VRs)
- We need to talk about scopes

Class-IR

Inputs/outputs (IO): 32-bit typed inputs

e.g.: `int x, int y, float z`

Program Variables (Variables): 32-bit untyped virtual register

given as `vrX` where `X` is an integer:

e.g. `vr0, vr1, vr2, vr3` …

we will assume input/output names are disjoint from virtual register names

Two different ID nodes

Gets compiled into an untyped virtual register

```
class ASTVarIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.node_type = value_type
```

Gets compiled into a typed IO variable

```
class ASTIOIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.node_type = value_type
```

Two different ID nodes

What we are compiling

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = i;
  }
}
```

Class-IR

What we are compiling

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = i;
  }
}
```

IO variables

program variables

```
int main() {
  int a = 0;
  test1(a);
  cout << a << endl;
  return 0;
}
```

What does this print?

What we are compiling

IO variables

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = i;
  }
}
```

program variables

Every time you access an IO variable, you need to convert it to a vr first using float2vr or int2vr

```
class ASTIOIDNode(ASTLeafNode):
...
    def three_addr_code(self):
        if self.node_type == Types.INT:
            return "%s = int2vr(%s);" % (self.vr, self.value)
        if self.node_type == Types.FLOAT:
            return "%s = float2vr(%s);" % (self.vr, self.value)
```

What we are compiling

IO variables

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = i;
  }
}
```

program variables

Every time you access a program variable, it does not need to be converted.

Because its value is a virtual register, you can even just use its value as its virtual register

```
class ASTVarIDNode(ASTLeafNode):
...

  def three_addr_code(self):
    return "%s = %s;" % (self.vr, self.value)
```

building an expression AST, we parse a unit at the base

```
unit := ID
     |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match[1]
   data_type = # get type from symbol table
   eat("ID")
   return ASTIDNode(id_name, data_type)
}
```

Previously we had just one ID node

building an expression AST, we parse a unit at the base

```
unit := ID
      |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match[1]
   data_type = # get type from symbol table
   eat("ID")
   return ASTIDNode(id_name, data_type)
}
```

building an expression AST, we parse a unit at the base

```
unit := ID
      |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match[1]
   id_data = # get id_data from the symbol table
   eat("ID")
   return ASTIDNode(id_name, ...)
}
```

id_data should contain:
***`id_type`**: IO or Var*
***`data_type`**: int or float*

building an expression AST, we parse a unit at the base

```
unit := ID
     |   ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match[1]
   id_data = # get id_data from the symbol table
   eat("ID")
   if (id_data.id_type == IO)
       return ASTIOIDNode(id_name, id_data.data_type)
   else
       return ASTVarIDNode(id_name, id_data.data_type)
}
```

id_data should contain:
`id_type`*: IO or Var*
`data_type`*: int or float*

Getting back to our statements:

```
statement := declaration_statement
          |  assignment_statement
          |  if_else_statement
          |  block_statement
          |  for_loop_statement
```

When we declare a variable, we need to mark it as a program variable in the symbol table

Getting back to our statements:

```
statement := declaration_statement
          |  assignment_statement
          |  if_else_statement
          |  block_statement
          |  for_loop_statement
```

We need to use symbol table data for something else. What?

Getting back to our statements:

```
statement := declaration_statement
          |  assignment_statement
          |  if_else_statement
          |  block_statement
          |  for_loop_statement
```

We need to use symbol table data for something else. What?

Scopes! Class IR has no {}s, so we need to manage scopes

Scopes

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

What does y hold?

Scopes

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

How can we get rid of the {}'s?

What does y hold?

Scopes

Let's walk through it with a symbol table

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

Scopes

Let's walk through it with a symbol table

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

symbol table hash table stack

Scopes

rename

Let's walk through it with a symbol table

```
int x_0;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

make a new unique name for x

HT0

x: (INT, VAR, "x_0")

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

x: (INT, VAR, “x_0”)

symbol table hash table stack

Scopes

rename

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

make a new unique name for y

HT0

```
x: (INT, VAR, “x_0”)
y: (INT, VAR, “y_0”)
```

symbol table hash table stack

Scopes

search

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

replace
with
new name

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

x: (INT, VAR, “x_0”)
y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x;
  x = 6;
  y = x;
}
```

new scope. Add x with a new name

HT1	x: (INT, VAR, "x_1")
HT0	x: (INT, VAR, "x_0") y: (INT, VAR, "y_0")

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x = 6;
  y = x;
}
```

new scope. Add x with a new name

HT1	`x: (INT, VAR, "x_1")`
HT0	`x: (INT, VAR, "x_0")` `y: (INT, VAR, "y_0")`

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1

```
x: (INT, VAR, "x_1")
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1

```
x: (INT, VAR, "x_1")
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
 int x_1;
 x_1 = 6;
 y_0 = x_1;
}
```

lookup

new scope. Add x with a new name

HT1

```
x: (INT, VAR, "x_1")
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y_0 = x_1;
}
```

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

No more need for {}

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
int x_1;
x_1 = 6;
y_0 = x_1;
```

new scope. Add x with a new name

HT1	x: (INT, VAR, "x_1")
HT0	x: (INT, VAR, "x_0") y: (INT, VAR, "y_0")

symbol table hash table stack

No more need for {}

Scopes

What happens with multiple scopes?

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
}
{
  int x;
  x = 1;
  y = x;
}
```

Scopes

What happens with multiple scopes?

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
}
{
  int x;
  x = 1;
  y = x;
}
```

What if x is uninitialized?

Class-IR

Remind ourselves what we are compiling

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = x + i;
  }
}
```

We only need new names for program variables, not for IO variables

building an expression AST, we parse a unit at the base

```
unit := ID
     |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match[1]
   id_data = # get id_data from the symbol table
   eat("ID")
   if (id_data.id_type == IO)
       return ASTIOIDNode(id_name, id_data.data_type)
   else
       return ASTVarIDNode(id_data.new_name, id_data.data_type)
}
```

id_data should contain:
***id_type**: IO or Var*
***data_type**: int or float*
***new_name**: new unique name*

NEXT:

- Finish up talking about intermediate representaitons