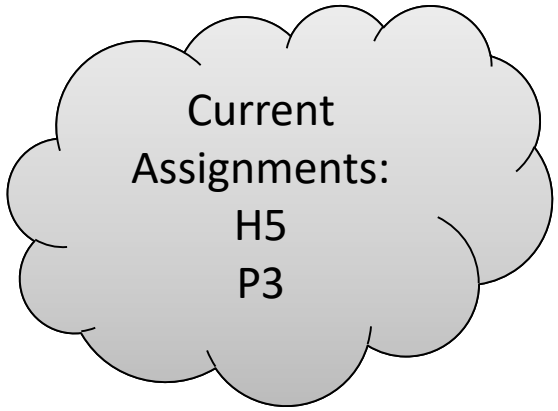
KU | Fall 2018 | Drew Davidson

EECS 665 COMPILER CONSTRUCTION

20 – Intro to Types

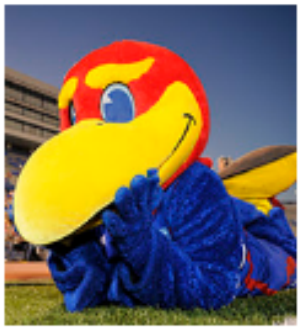
Administrivia



- Turn in H5
- H6 will be out by 5:00 PM
  - Will send a note on Blackboard

Administrivia

Current Assignments:
H5
P3



Future Jayhawks visiting for Crimson & Blue Day

Oct. 12

Prospective students and their families will visit the Lawrence campus on Friday for an all-university open house — a great opportunity for faculty and staff say hello and answer questions.

Read More »

These efforts are always ongoing, of course. Next week, we will have a special opportunity to welcome prospective students and families to Lawrence for our annual Crimson & Blue Day. On Friday, Oct. 12, high school students, transfer students and their families will participate in campus tours, academic sessions, resource fairs and other customized appointments to help them explore KU. More information, including a full schedule of events, is available at admissions.ku.edu/crimsonandblue.

....Yaaay

Lecture Outline

- Last time
  - Consider an instance of semantic analysis: name analysis (AKA name resolution)
- This time
  - Types
    - Common typing design points
    - Our type rules (and how to apply them)

Say what *is* a Type?

- Short for “data type”
  - Classification identifying kinds of data
  - A set of possible values which a variable can possess
  - Operations that can be done on member values
  - A representation (perhaps in memory)

Components of a Type System

- Primititve types + means of building aggregate types
  - Int, bool, void, class, function, struct
- A means of dedetermining if types are compatible
  - Can disparate types be combined? How?
- Rules for inferring the type of an expression

Type Rules

- For every operator (including assignment)…
  - What types can the operand have?
  - What type is the result?
- Example:

```
double a;
int b;
a = b;
b = a;
```

Legal in Java, C++

Legal in C++, Illegal in Java

Type Coercion

- Implicit cast from one data type to another
  - float to int
  - Narrow Form: type promotion
    - When the destination type can represent the source type
    - float to double

Types of Typing: When do we check?

- Static typing
  - Type checks are made before execution of the program (compile-time)
- Dynamic typing
  - Type checks are made during execution
- Combination of the two
  - Java (downcasting vs cross-casting)

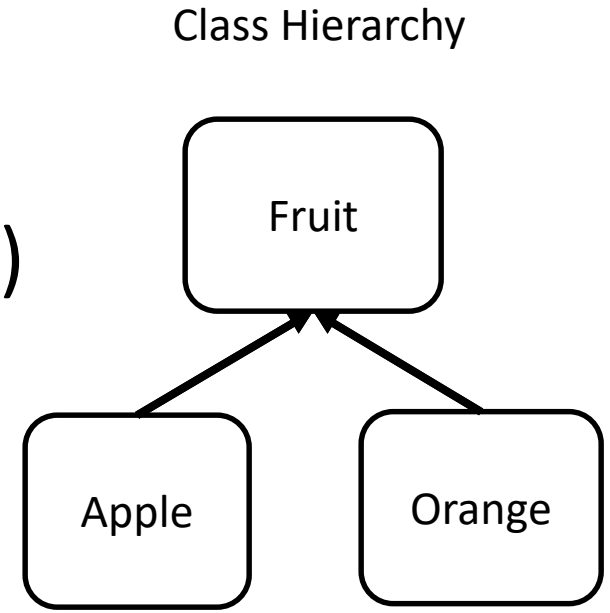
Typing Example: Casting

- Cross-casting (static check)

```
Apple a = new Apple();
Orange o = (Orange)a;
```

- Downcasting (dynamic check)

```
Fruit f = new Apple();
if ( … ) {
  f = new Orange();
}
Apple dApp = (Apple)f;
```



Static v Dynamic Benefits

- Statically typed
  - Compile-time error reports
  - Compile-time optimization
- Dynamically typed
  - Avoid dealing with errors that don't matter
  - Some added flexibility

Duck Typing

- Type is defined by the methods and properties

"If it walks like a duck and talks like a duck, it's a duck"

Duck Typing: Example

```
class Duck:
    def quack(): print(“quack”)
class Rando:
    def quack(): print(“QUACK”)

function processDuck(Duck d) { … }
Rando r = new Rando();
processDuck(r);
```

Duck Punching

- Type is defined by the methods and properties

"If it walks like a duck but isn't giving you the noise you want, punch it until it quacks. Now it's a duck"

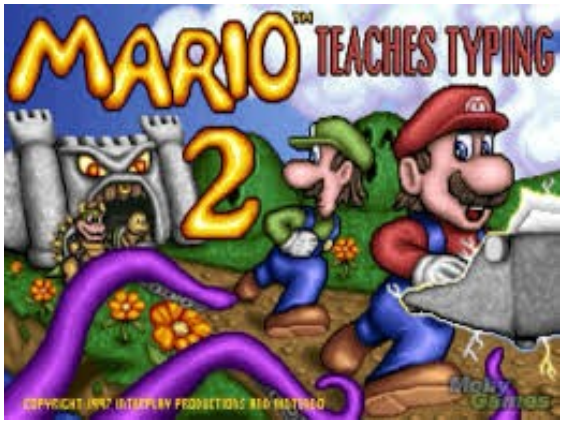
Duck Punching: Example

```
class Duck:
    def quack(): print(“quack”)
class MechaBird:
    def squak(): print(“101001…”)

function processDuck(Duck d) { … }
MechaBird m = new MechaBird();
m.quack = m.squak;
processDuck(m);
```

Types of Typing, cont: What do we check?

- Strong vs weak typing
  - Degree to which type checks are performed
  - Degree to which type errors are allowed to happen at runtime
  - Continuum without precise definitions



Type Safety

- Has a precise definition
  - All successful operations must be allowed by the type system
- Java was explicitly designed to be type safe
  - A variable of some type can only be used as that type without causing an error
- C is very much not type safe
- C++ isn't either but it is safer

Type Safety Violations

C

Format specifier

```
printf(“%s”, 1);
```

Memory safety

```
Struct big{
  int a[1000000];
};
Struct big * b = malloc(1);
```

C++

Unchecked casts

```
class T1 { char a };
class T2 { int b };
int main {
  T1 * myT1 = new T1();
  T2 * myT2 = new T2();
  myT1 = (T1*)myT2;
}
```

Let’s Talk about Lil’ C

Our Type System

- Primitive Types
  - int, bool, string, void
- Aggregate types
  - struct, functions
- Coercion
  - Bool cannot be used as an int (nor vice-versa)

C-Flat Type Errors (1)

- Arithmetic operators must have **int** operands
- Equality operators **==** and **!=**
  - Operands must have same type
    - Cant' be applied to
      - Functions (but CAN be applied to function results)
      - Struct names
      - Struct variables
- Other relational operators must have **int** operands
- Logical operators must have **bool** operands

C-Flat Type Errors (2)

- Assignment operator
  - Must have operands of the same type
  - Can't be applied to functions
    - Functions (but CAN be applied to function results)
    - Struct name
    - Struct variables
- For input >> x;
  - x cannot be function, struct name, struct variable
- For output << x;
  - X cannot be function, struct name, struct variable
- Condition of **if** and condition of **while** must be boolean

C-Flat Type Errors (3)

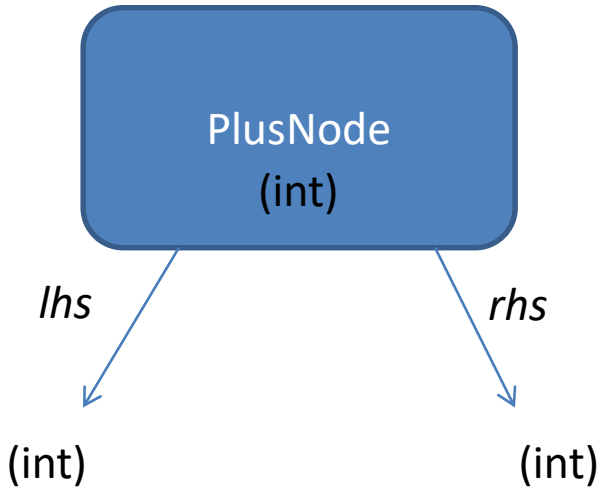
- Invoking (calling) something that's not a function
- Invoking a function with
  - Wrong number of args
  - Wrong type of args
    - Also will disallow structs or functions as args
- Returning a value from a void function
- Not returning a value in a non-void function
- Returning a wrong type of value in a non-void function

Type Checking

- Structurally similar to nameAnalysis
  - Historically, intermingled with nameAnalysis and done as part of attribute “decoration”
- Add a typeCheck method to AST nodes
  - Recursively walk the AST checking subtypes
  - Let’s look at a couple of examples

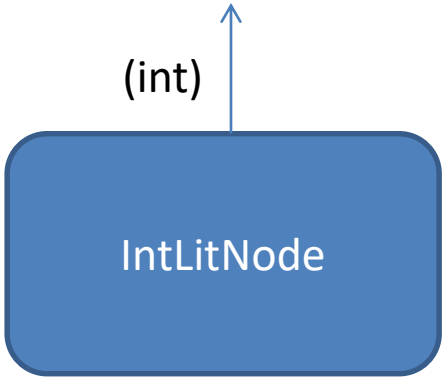
Type Checking: Binary Operator

- Get the type of the LHS
- Get the type of the RHS
- Check that the types are compatible for the operator
- Set the *kind* of the node be a value
- Set the *type* of the node to be the type of the operation's result



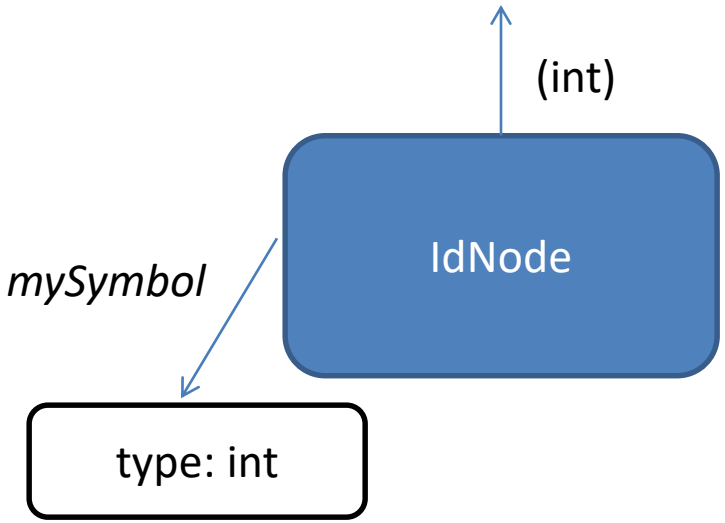
Type "Checking" Literals

- Cannot be wrong
  - Just pass the type of the literal up the tree



Type Checking: IdNode

- Look up the type of the declaration
  - There should be a symbol “linked” to the node
- Pass symbol type up the tree



Type Checking: Others

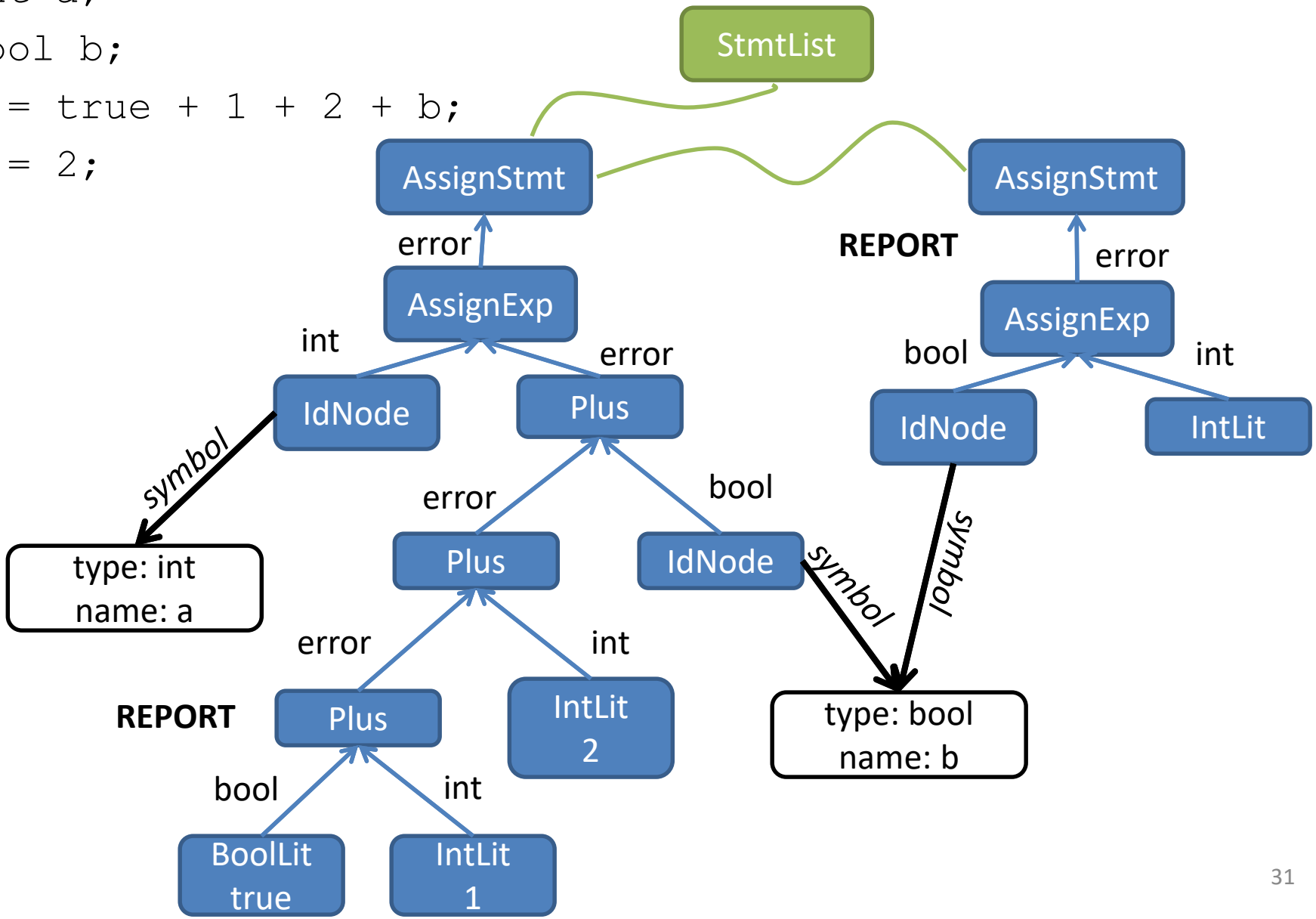
- Other node types follow these same principles
  - Function calls
    - Get type of each actual argument
      - Match against the formal argument (check symbol)
    - Send the return type up the tree
  - Statement
    - No type

Type Checking: Error Reports

- We'd like all *distinct* errors at the same time
  - Don't give up at the first error
  - Don't report the same error multiple times
- Introduce an internal **error** type
  - When type incompatibility is discovered
    - Report the error
    - Pass **error** up the tree
  - When you get error as an operand
    - Don't (re)report an error
    - Again, pass **error** up the tree

Type Error Example

```
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```



Next Time

- We'll talk about how we perform type checking
  - Compile-time type checking in the compiler
  - Inserting dynamic checks in the compiler's output