Types and Type Checking

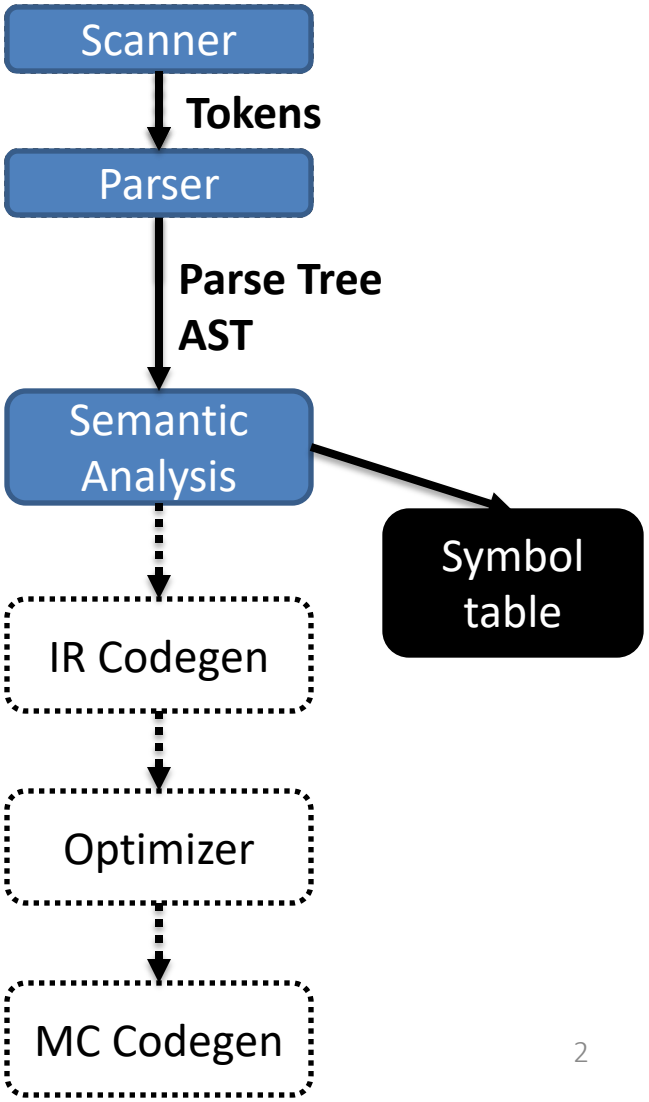
Roadmap

Name analysis

- Static scoping
- Tree traversal, with symbol-table operations (new, insert, lookup)

Today

- Type checking



Lecture Outline

Type Safari

- Type-system concepts
- Type-system vocabulary

b

- Type rules
- How to apply type rules

Data representation

- Moving towards actual code generation
- Brief comments about types in memory

Say, What *is* a Type?

Short for “data type”

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

Type Intuition

The language does not allow you to do the following:

```
int a = 0;
int * pointer  = &a;
float fraction = 1.2;
a = pointer + fraction;
```

… or does it?

Components of a Type System

Primitive types + operators for building more complex types

- int, bool, void, class, function, struct

Means of determining if types are compatible

- Can values with different types be combined?
- If so, 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?

Examples

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

Legal in Java, C++

Legal in C++, not 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 I: **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 (runtime)

Combination of the two

- Java (downcasting vs cross-casting)

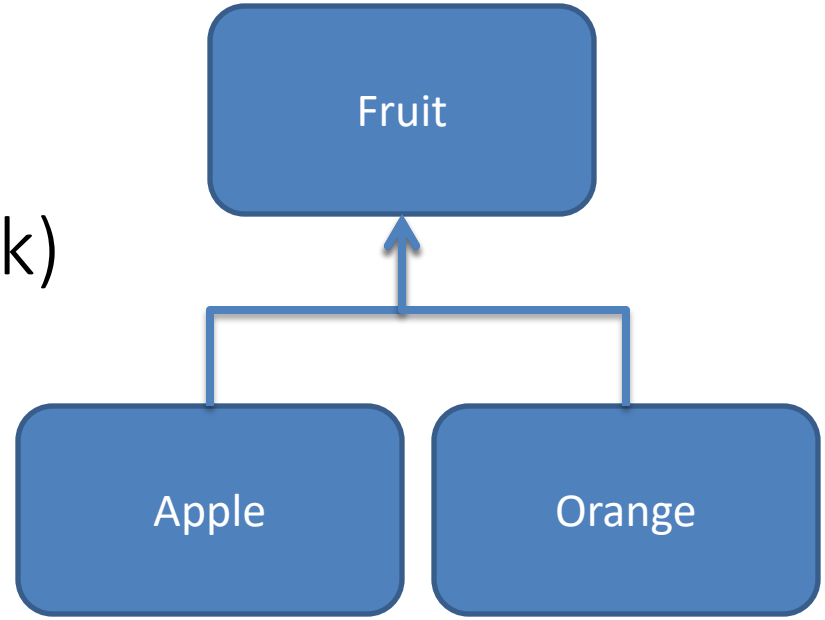
Example: Casting

Cross-casting (static check)

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

Downcasting (dynamic check)

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



Static vs. Dynamic Tradeoffs

Statically typed

- Compile-time optimization
- Compile-time error checking

Dynamically typed

- Avoid dealing with errors that don't matter
- Some added flexibility
- Failures can happen at runtime
  - . . . in a fielded product
  - Test suites rarely exercise all code under all different runtime situations

Duck Typing

Type is defined by the methods and properties

```
class bird:
    def quack(): print("quack!")
class mechaBird:
    def quack(): print("101011…")
```

How do we arrange it?

– (Some languages) "Duck punching": Runtime modification of object to add an additional method

Types of Typing II: **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

Strong vs. Weak

No universal definitions but …

- Statically typed is often considered stronger (fewer type errors possible)
- The more implicit casts allowed the weaker the type system
- The fewer checks performed at runtime the weaker the type system

Strong vs. Weak Example

C (weaker)

```
union either{
    int i;
    float f;
} u;
u.i = 12;
float val = u.f;
```

StandardML (stronger)

```
real(2) + 2.0
```

Fancier types

Dependent types can be used to reason about computation

- Reverse takes a list of int of length n and returns a list of int of length n

Resource types can be used to reason about program complexity

- The program only type-checks if it runs in poly time

Very hard to reason about, but strong guarantees

Type Safety

Type safety

- All successful operations must be allowed by the type system
- Java was explicitly designed to be type safe
  - If you have a variable with some type, it is guaranteed to be of that type
- C is not
- C++ is a little better

Computer scientist Ross Tate working to tame Java 'wildcards'

By Bill Steele

A Cornell computer scientist has just discovered that the Java computer language, designed to be safe, is not so safe after all, and now he is working to find a solution.

Type-Safety Violations

C

- Format specifier

```
printf("%s", 1);
```

- Memory safety

```
struct big{
    int a[100000];
};
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;
}
```

Type System of b

b

b's type system

Primitive types

- int, bool, string, void

Type constructors

- struct

Coercion

- bool cannot be used as an int in our language (nor vice-versa)

b Type Errors I

Arithmetic operators must have **int** operands

Equality operators **==** and **!=**

- Operands must have same type
- Can't be applied to
  - Functions (but CAN be applied to function results)
  - struct name
  - struct variables

Other relational operators must have **int** operands

Logical operators must have **bool** operands

b Type Errors II

Assignment operator

- Must have operands of the same type
- Can't be applied to
  - Functions (but CAN be applied to function results)
  - struct name
  - struct variables

For cin >> x;

- x cannot be function, struct name, struct variable

For cout << x;

- x cannot be function, struct name, struct variable

Condition of if, while must be bool

b Type Errors III

Invoking (a.k.a. calling) something that is not a function

Invoking a function with

- Wrong number of arguments
- Wrong types of arguments
  - Also will not allow structs or functions as arguments

Returning a value from a void function

Not returning a value in a non-void function function

Returning wrong type of value in a non-void function

Type Checking

Structurally similar to nameAnalysis

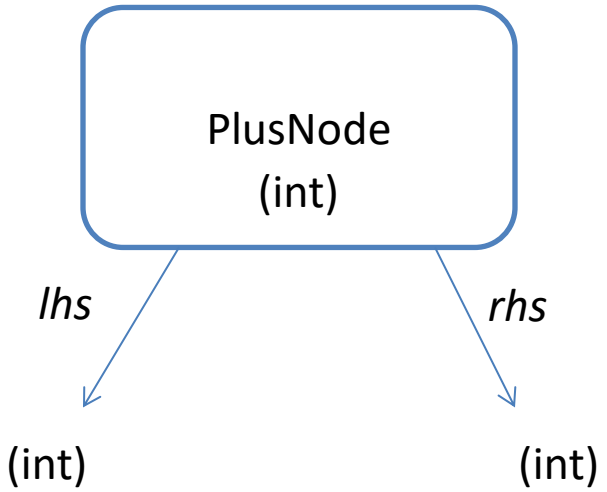
- Sometimes intermingled with nameAnalysis and done as part of attribute “decoration”
- Don’t do that . . .

Add a typeCheck method to AST nodes

- Recursively walk the AST checking types of sub-expressions
- Let’s look at a couple of examples

Type Checking: Binary Operator

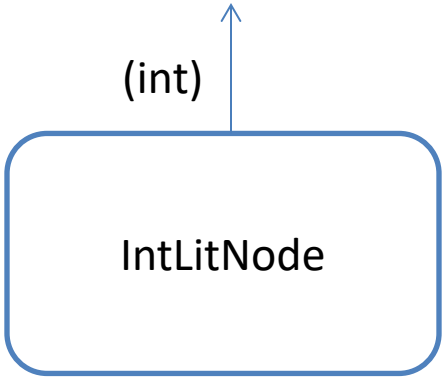
- Get the type of the LHS
- Get the type of the RHS
- Check that the types are compatible for the operator
- Set the *kind* of the node be a value
- Set the *type* of the node to be the type of the operation's result



Type “Checking”: Literal

Cannot be wrong

- Just pass the type of the literal up the tree

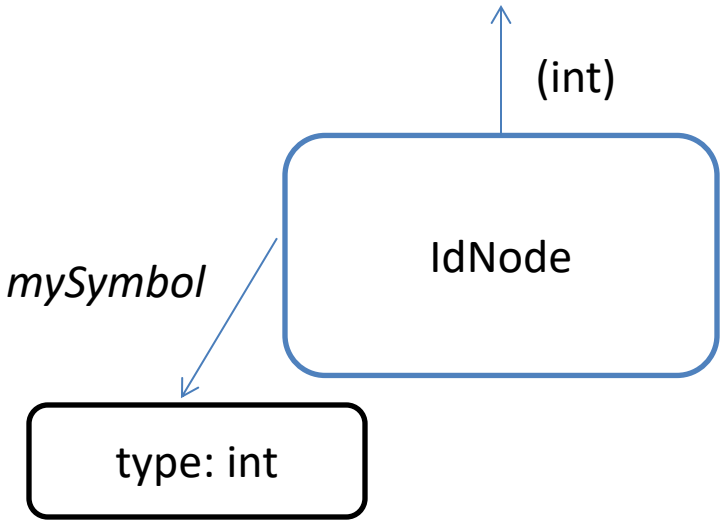


Type Checking: IdNode

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

- A call to function f
  - Get the type of each actual parameter of f
  - Match against the type of the corresponding formal parameter of f
    - use the information in the symbol-table entry for f
  - Pass f's return type up the tree
- Statement s
  - Type check the constituents of s
  - Nothing to pass up the tree: A statement does not produce a value, and hence s has no "return type"

Type Checking: Errors

Goals

- Report multiple errors
- Don't report the same error multiple times (i.e., avoid error cascading)

We'd like the compiler to report as many *distinct* errors as possible

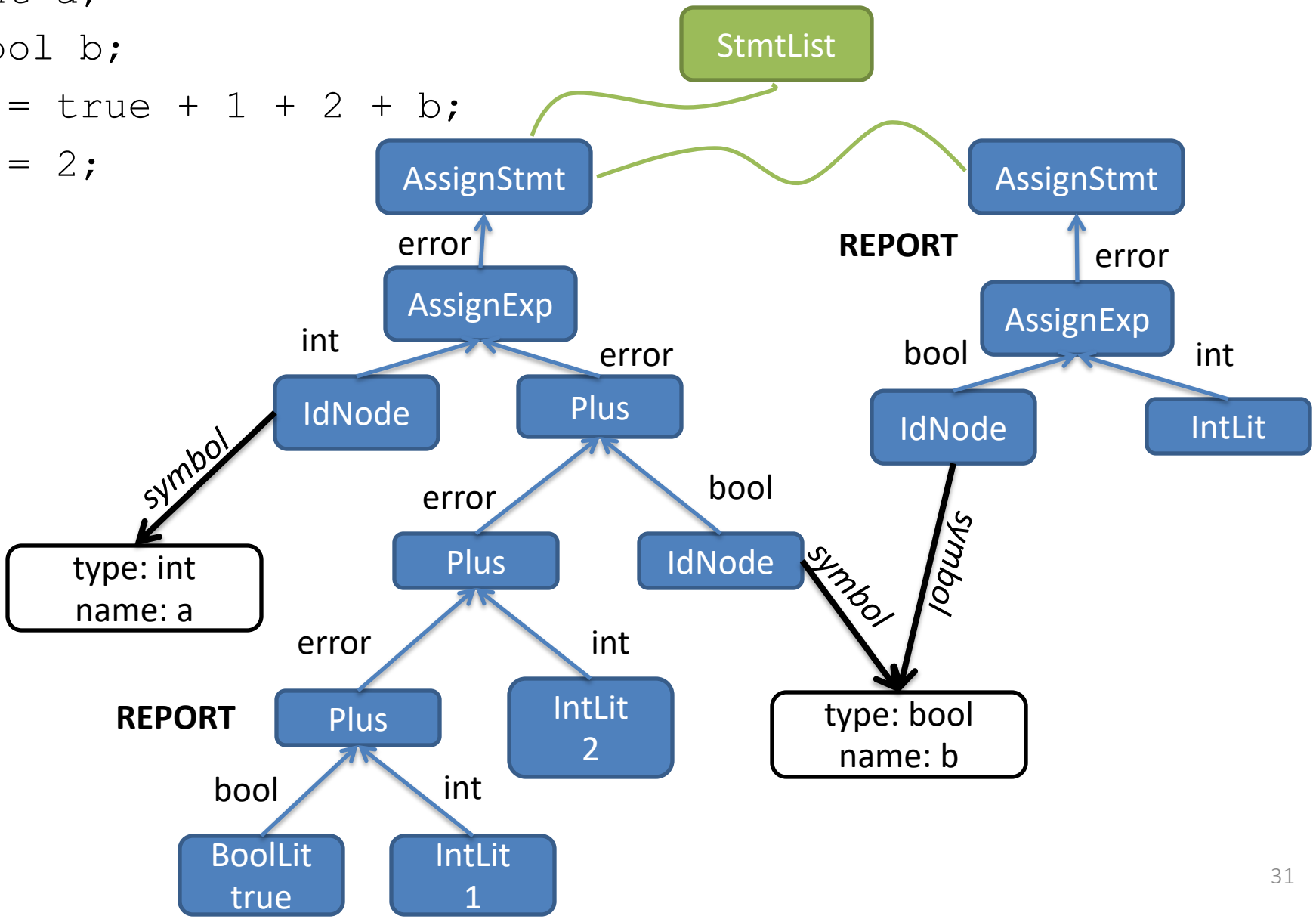
- It mustn't give up at the first error
- Internally, it needs to know if an error has already been reported

Introduce an internal **error** type

- When type incompatibility is discovered
  - Report the error
  - Pass **error** up the tree
- When a type check gets **error** as an operand
  - Don't (re)report an error
  - Again, pass **error** up the tree

Error Example

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



Looking Towards Next Lecture

- Look at how data (and therefore a value of some type) is represented in the machine
- Start very abstract; won't talk about an actual architecture for a while
- Assembly has no intrinsic notion of types. One would have to add code for checking types (if runtime checks are needed)