Types
Roadmap

• Back from our LR Parsing Detour
• Name analysis
  – Static v dynamic
  – Scope
• Today
  – Type checking
Lecture Outline

• Type Safari
  – Type system concepts
  – Type system vocabulary

• CATS
  – 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 which a variable can possess
  – Operations that can be done on member values
  – A representation (perhaps in memory)
Type Intuition

You can’t do this:

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

... or can you?
Components of a type system

• Primitive types + means of building aggregate types
  – int, bool, void, class, function, struct
• Means of determining if types are compatible
  – Can disparate types be combined? How?
• Rules for inferring type of an expression
Type Rules

• For every operator (including assignment)
  – What types can the operand have?
  – What type is the result?

• Examples

```java
double a;
int b;
a = b;  // Legal in Java, C++
b = a;  // 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 v 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 v Dynamic Tradeoffs

• Statically typed
  – Compile-time optimization
  – Compile-time error checking

• Dynamically typed
  – Avoid dealing with errors that don’t matter
  – Some added flexibility
Duck Typing

• Type is defined by the methods and properties

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

• Duck Punching
  – Runtime modifications to allow duck typing
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
Strong v Weak Example

• C (weaker)

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

• SML (stronger)

```sml
real(2) + 2.0
```
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
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 CATS

CATS
CATS 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)
CATS 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
CATS 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 boolean
CATS Type Errors III

• Invoking (aka calling) something that’s not a function

• Invoking a function with
  – Wrong number of args
  – Wrong types of args
    • Also will not allow struct or functions as args

• Returning a value from a void function

• Not returning a value in a non-void function

• Returning 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

```
PlusNode (int)
```

```
lhs (int)
```

```
rhs (int)
```
Type “Checking”: Literal

• Cannot be wrong
  – Just pass the type of the literal up the tree

```
Type "Checking": Literal

• Cannot be wrong
  – Just pass the type of the literal up the tree

25
```
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

```
IdNode

mySymbol

type: int

(int)
```
Type Checking: IdNode

- Look up the type of the declaration

---

PlusNode
(int)

lhs
(int)

rhs
(int)
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: Errors

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

Error Example

```
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```
Looking Towards Next Lecture

• Starting Thursday
  – Look at how data (and therefore types) is represented in the machine
  – Start very abstract, won’t talk about an actual architecture for awhile
  – Assembly has no intrinsic notion of types. We’ll have to add code for type checking ourselves