Prelude

- What is this spacecraft? *Mars Climate Orbiter*
- What happened to it?
  - After 268 day journey, it attempted to enter into Mars orbit
  - Passed too close, crashed
- Why?
  - Engineering team at Lockheed using *English units*, NASA using *metric units*

Beyond syntax

- What’s wrong with this code? *(Note: it parses perfectly)*

```c
foo(int a, char * s){ .. }

int bar() {
  int f[3];
  int i, j, k;
  char *p;
  float x;
  foo(f[6], 10, j);
  break;
  i->val = 5;
  j = i + k;
  printf("%s,%s.
",p,q);
  goto label23;
}
```

Errors

- Undeclared identifier
- Multiply declared identifier
- Index out of bounds
- Wrong number or types of args to call
- Incompatible types for operation
- Break statement outside switch/loop
- Goto with no label

Kinds of checks

- Uniqueness checks
  - Certain names must be unique
  - Many languages require variable declarations
- Flow-of-control checks
  - Match control-flow operators with structures
  - Example: break applies to innermost loop/switch
- Type checks
  - Check compatibility of operators and operands

Static checks

Why do we care?

- Obvious: report mistakes to programmer
  - *f[6]* will cause a run-time failure
  - Help programmer verify intent
- How do these checks help compiler?
  - Allocate right amount of space for variables
  - Select right machine operations
  - Proper implementation of control structures
Today

Focus on type systems
- Type system
  - Type expressions
  - Type inference
- Checking and conversions
  - Static vs dynamic checks
  - Type conversions
- Polymorphism

Other semantic checks
- Based on symbol tables
  - Add entries to symbol table
  - Check references against table
  - Example: declarations
    - Add each declaration to table
    - Make sure entries are unique
    - Check that variable uses refer to entries in table
- Closely tied to notion of scope
  We’ll address these topics when we discuss procedures

Types
- Duff’s device

Types
- As programmers...
  - We have an intuitive notion of types
  - What is a type?

Types
- From Types and Programming Languages
  “A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.”

  - Idea
    - Divide possible values into groups – types
    - Disallow certain behaviors based on membership

Types and compilers
- Compiler must understand the type system of the language
  - Enforce limitations
  - Implement checks and conversions
  - Provide concrete representation – bits
- Responsibilities vary by language
  - C/C++ – all type checking occurs in compiler
  - Java – mixed model
  - Perl, Ruby – no compiler, all dynamic checking
**Type systems**

From language specifications:

“The result of a unary & operator is a pointer to the object referred to by the operand. If the type of the operand is "T", the type of the result is "pointer to T".

“If both operands of the arithmetic operators addition, subtraction and multiplication are integers, then the result is an integer”

**Properties of types**

What do these excerpts imply?

- Types have structure
  - “Pointer to T” and “Array of Pointer to T”
- Expressions have types
  - Types are derived from operands by rules
- Goal: determine types for all parts of a program

**Type expressions**

(Not to be confused with types of expressions)

- Build a description of a type from:
  - Basic types – also called “primitive types”
    - Vary between languages: int, char, float, double
  - Type names
    - An “alias” for a type expression – typedef in C
  - Type variables
    - Unspecified parts of a type – polymorphism
  - Type constructors
    - Functions over types that build more complex types

**Type constructors**

- Arrays
  - If T is a type, then array(T) is a type denoting an array with elements of type T
  - May have a size component: array(I,T)
- Products and records
  - If T1 and T2 are types, then T1×T2 is a type denoting pairs of two types
  - May have labels for records/structs (“name”, char *) × (“age”, int)

**Graphically**

- View types as graphs
  - Node for each type
  - Often created as a DAG for efficiency

- Example graph:
  - Node for int, char,_pointer
  - Edges: "char" → "int" → "pointer"

Function: (char × int) → int *
Type checking
- Define language type system
  - Set of rules for assigning type expressions to the parts of a program
- Implemented by type checker
  - Derives types using rules
  - Static (compile-time) or dynamic (run-time)
- Type checking may fail
  - Handling errors depends on specific constructs

Example
- Static type checker for C
  - Assume:
    We can get declared types of identifiers, functions
  - Rules:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₁ ( E₂ )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if type(E₁) is D ( \rightarrow ) R and type(E₂) is D</td>
</tr>
<tr>
<td></td>
<td>result type is R</td>
</tr>
<tr>
<td></td>
<td>else error</td>
</tr>
</tbody>
</table>

* E
  - if type(E) is pointer(T)
  - result type is T
  - else error

Example
- What about function calls?
  - Consider single argument case
  - Extends to multiple arguments (products)
  - What is fundamental operation?
    "If two type expressions are equivalent then..."

Type equivalence
- Implementation: structural equivalence
  - Same basic types
  - Same set of constructors applied
  - Recursive test:
    function equiv(s, t)
    if s and t are the same basic type
      return true
    if s = pointer(s₁) and t = pointer(t₁)
      return equiv(s₁, t₁)
    if s = s₁×s₂ and t = t₁×t₂
      return equiv(s₁, t₁) \&\& equiv(s₂, t₂)
    ...etc...

Efficiency is critical

Name equivalence
- Different way of handling type names
  - Structural equivalence
    - Ignores type names
    - typedef int * numptr \rightarrow numptr \equiv int *
    - Not always desirable
  - Name equivalence
    - Types are equivalent if they have the same name
    - Solves an important problem: recursive types
Recursive types

- Cycle in the type graph:
  ```
  struct cell {
    int info;
    struct cell * next;
  }
  ```
- C uses structural equivalence for everything except structs
  - The name “struct cell” is used instead of checking the actual fields in the struct
  - Can we have two compatible struct definitions?

Type equivalence for Java

Type checking

- Consider this case:
  What is the type of \(x + i\) if \(x\) is float and \(i\) is int
- Is this an error?
- Compiler fixes the problem
  - Convert into compatible types
  - Automatic conversions are called **coercions**
  - Rules can be complex
    - in C, large set of rules for called **integral promotions**
    - Goal is to preserve information

Type coercions

- Rules
  - Find a common type
  - Add explicit conversion into the AST
<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
</table>
  | \(E_1 \times E_2\) | if type(E₁) is int and type(E₂) is int result type is int
  |              | if type(E₁) is int and type(E₂) is float result type is float
  |              | if type(E₁) is float and type(E₂) is int result type is float
  |              | etc... |

Overloading

- “+” operator
  - Same syntax, multiple implementations
  - C: float versus int
  - C++: arbitrary user implementation
- How to decide which one?
  - Use types of the operands
  - Find operator with the right type signature
- How does this interact with coercions?

Object oriented types

- What is relationship between \(foo\) and \(bar\)?
  - Any code that accepts a \(foo\) object can also accept a \(bar\) object
  - \(bar\) is a subtype of \(foo\)
- Modify type compatibility rules
  - Formal parameter can accept any subtype
  - Same holds for assignment
Type inference

- Languages without declarations
  - ML, Haskell
  - Still statically typed
  - Types determined by use
- Requires type inference algorithm
  - Determine constraints from program
  - Results in type expressions with type variables
  - Compute a consistent assignment to variables

Implementing type checkers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E₁ [ E₂ ]</td>
<td>If type(E₂) is int and type(E₁) is array(T) type(E) = T else error</td>
</tr>
<tr>
<td>E \rightarrow * E</td>
<td>If type(E) is pointer(T) type(E) = T else error</td>
</tr>
</tbody>
</table>

- Does this form look familiar?
  - Type checking fits into syntax-directed translation
  - One reason why declarations precede stmts

Expressions

- Why compute types of all expressions?
  - Example: x + y + z
    what is type of (x+y)?
- Already mentioned
  - Check for correctness
  - Add coercions
- Code generation
  - Most machines cannot add 3 values
  - Must break up the expression
  - Store intermediate results

Interesting cases

- What about printf?
  - printf(const char * format, …)
  - Implemented with varargs
  - Format specifies which arguments should follow
  - Who checks?
- Array bounds
  - Array sizes rarely provided in declaration
  - Cannot check statically

Polymorphism

- Ordinary procedures
  - Accept fixed type signature
  - Example: search(char c, string s)
- Generic procedures
  - Work on arguments of different types
  - Example: arithmetic operators (most languages)
- User-defined generics
  - Define interface with type variables
  - Example: search(El c, El [] list)

Polymorphic type checking

- Problem:
  - Is search(5, A) correct? (Let's say int A[20])
  - Find a mapping from 5 and A to El and El[]
  - Mapping: El is int
  - This process is called unification
- Unification
  - Given two type expressions, one with type variables
  - Find a substitution of type variables that turns it into the other type expression
  - Used in many other areas: theorem proving, Prolog
Generics in Java

- New in Java 1.5
  - Type variables, like C++ templates
  - Not as powerful (on purpose)
    boolean search(T el, List<T> list)

- Question:
  - How is this different from:
    boolean search(Object el, List list)
  (Where list is a linked list of Object references)

Back to Mars Orbiter

- Language support for units
  - Idea: make units a part of the type
  - Use polymorphism for operators
  - Type-check the computations

- Example:
  double<kg> weight;
  double<ms> time;
  double<m> distance;
  double<m s^-2> gravity = 9.8 * m/s * s;
  double<kg m s^-2> force = weight * gravity;

- Issues:
  - Type checker must understand algebra
  - Generics are tricky
    (what is the type of sqrt?)

Next time...

- The procedure abstraction
  - Symbol tables and scopes
  - Run-time environments
  - Storage allocation, stacks

- Later today: new programming assignment