**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*

---

**Where are we...**

- Parsing complete  
  - Syntax is correct  
  - Built an internal representation  
    (usually an abstract syntax tree)  
  - Now what?

**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 q, *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 5;
  j = i + k;
  printf("%s,%s\n",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

**Program checking**

*Why do we care?*

- Obvious:  
  - Report mistakes to programmer  
  - Avoid bugs:  *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
Kinds of static 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
  - Example: does 3.5 * "foobar" make sense?
- What kind of check is "array bounds"?

Procedures

- What is a procedure/function/method?
- Does it exist at the machine code level?
  - Not really – it’s an abstraction created by the compiler
  - Components
    - Name space abstraction
    - Control abstraction
    - Interface
- Today: name space abstraction
  - Defines scoping and binding rules
- Later: look at how abstraction is implemented
Procedure: three abstractions

- **Control abstraction**
  - Well defined entries & exits
  - Mechanism to return control to caller
  - Some notion of parameterization (usually)

- **Clean name space**
  - Clean slate for writing locally visible names
  - Local names may obscure identical, non-local names
  - Local names cannot be seen outside

- **External interface**
  - Access is by procedure name & parameters
  - Clear protection for both caller & callee
  - Invoked procedure can ignore calling context

Procedures as name spaces

Each procedure creates its own name space

- Any name (almost) can be declared locally
- Local names hide identical non-local names (shadowing)
- Local names cannot be seen outside the procedure
- We call this set of rules & conventions **lexical scoping**
- Scopes may be nested

Examples

- C has global, static, local, and block scopes
- Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes
- Procedure scope (typically) contains formal parameters

Procedures as name spaces

- **Why introduce lexical scoping?**
  - Flexibility for programmer
  - Simplifies rules for naming & resolves conflicts

- **Implementation:**
  - The compiler responsibilities:
    - At point \(p\), which "x" is the programmer talking about?
    - At run-time, where is the value of \(x\) found in memory?

- **Solution:**
  - Lexically scoped symbol tables

Examples

- **In C++ and Java**

  ```java
  for (int i=0; i < 100; i++) {
     ...
  }
  for (Iterator i=list.iterator(); i.hasNext();) {
     ...
  }
  ```

- This is actually useful!

Lexically-scoped Symbol Tables

- **Compiler job**
  - Keep track of names (identifiers)
  - At a use of a name, find its information (like what?)

- **The problem**
  - Compiler needs a distinct entry for each declaration
  - Nested lexical scopes admit duplicate declarations

- **The symbol table interface**
  - `enter()` – enter a new scope level
  - `insert(name)` – creates entry for name in current scope
  - `lookup(name)` – lookup a name, return an entry
  - `exit()` – leave scope, remove all names declared there

Dynamic vs static

- **Static scoping**
  - Most compiled languages – C, C++, Java, Fortran
  - Scopes only exist at **compile-time**
  - We’ll see the corresponding **run-time** structures that are used to establish addressability later.

- **Dynamic scoping**
  - Interpreted languages – Perl, Common Lisp

```c
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```
**Example**

```java
class p {
    int a, b, c
    method q {
        int v, b, x, w
        for (r = 0; ...) {
            int x, y, z
            ...
        }
        while (s) {
            int x, a, v
            ...
        }
        ...
    }
}
```

**Chained implementation**

- Create a new table for each scope, chain them together for lookup

**Stack implementation**

- **Implementation**
  - `enter()` puts a marker in stack
  - `insert()` inserts at `nextFree`
  - `lookup()` searches from `nextFree-1` forward
  - `exit()` sets `nextFree` back to the previous marker.

  **Advantage**
  - Uses less space
  - Lookups can be expensive

- **Threaded stack implementation**

  - **Implementation**
    - `insert()` puts new entry at the head of the list for the name
    - `lookup()` goes direct to location
    - `exit()` processes each element in level being deleted to remove from head of list

  **Advantage**
  - Lookup is fast
  - `exit` takes time proportional to number of declared variables in level

**Symbol tables in C**

- **Identifiers**
  - Mapping from names to declarations
  - Fully nested – each `{` opens new scope

- **Labels**
  - Mapping from names to labels (for goto)
  - Flat table – one set of labels for each procedure

- **Tags**
  - Mapping from names to struct definitions
  - Fully nested

- **externals**
  - Record of extern declarations
  - Flat table – redundant extern declarations must be identical

  In general, rules can be very subtle

**Examples**

- **Example of typedef use:**
  ```c
  typedef int T;
  struct S { T T; }; /* redefinition of T as member name */
  ```

- **Example of proper declaration binding:**
  ```c
  int; /* syntax error: Vacuous declaration */
  struct S; /* no error; tag is defined, not elaborated */
  ```

- **Example of declaration name spaces**
  - Declare "a" in the name space before parsing initializer
    ```c
    int w = sizeof(a);
    ```
  - Declare "b" with a type before parsing "c"
    ```c
    int b, sizeof(b);
    ```
Uniqueness checks

- Which ones involve uniqueness?
- What do we need to do to detect them?

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