Handling Recursion

- Java, Iota: all global identifiers visible throughout their module (even before defn.)
- Need to create environment (symbol table) containing all of them for checking each function definition
- Global identifiers bound to their types
  \[ x : \text{int} \Rightarrow \{ \ldots x : \text{int} \ldots \} \]
- Functions bound to function types
  \[ \text{gcd}(x : \text{int}, y : \text{int}) : \text{int} \Rightarrow \{ \ldots \text{gcd} : \text{int} \times \text{int} \rightarrow \text{int} \ldots \} \]
- Auxiliary environment info
  - Entries representing functions are not normal environment entries
    \[ \{ \text{gcd} : \text{int} \Rightarrow \text{int} \} \]
  - Functions not first-class values in Iota: can't use gcd as a variable name
  - Need to flag symbol table entries
  - Other entries (return, etc.) also must be flagged

Recursive Types

- Type declarations may be recursive too
  - Java: `class List { Object head; List tail; }`
  - C: `struct GraphNode { struct Arc *firstArc; }`
  - `struct Arc { struct GraphNode *from, *to; struct Arc *nextArc; }

\[ \text{name: "GraphNode" fields: 0: "firstArc", } \]
\[ \text{name: "Arc" fields: 0: "from", 1: "to", 2: "nextArc", } \]
Interpreting type expressions
- How to convert recursive type expressions into cyclical graph structure?
- Solution: more semantic analysis passes
  - First pass: pick up all type names, create placeholder type objects and put into symbol table
  - Second pass: fill in type objects using symbol table to look up type names (can do global variables too)
  - Third pass: typecheck actual code
- Mantra #2: add another pass

Where we are
- Source code (character stream)
- Token stream
- Abstract syntax tree
- Abstract syntax tree + type objects, symbol tables

Our Intermediate Code
- Code for an abstract processor (in tree form)
- Processor-specific details avoided (e.g. # of registers)
- Generality enables (some) optimizations
- Conversion between tree representations

Variables in IR
- Variables mapped to memory locations
  - b \Rightarrow MEM[fp - 8]
- This lecture: how do we map them?

IR Architecture
- Infinite no. of general purpose registers
- Stack pointer register (sp)
- Frame pointer register (fp)
- Program counter register (pc)
- Versus Pentium:
  - Very finite number of registers (EAX–EDX, ESI, EDI)
  - None really “general purpose”
  - Stack pointer (ESP), frame pointer (EBP), instruction pointer (EIP)
- Versus MIPS, Alpha: 32 general purpose regs

Representing variables
- Global variables: mapped to particular locations in memory
- Local variables, arguments: can’t map to fixed locations because of recursion, threads
  
  ```
  fact(x: int): int = {
    if (x==0) 1; else x * fact(x-1);
  }
  ```
  
  where to store x?
  FORTRAN: stored in fixed memory locn!
**Stack**
- Local storage allocated on stack
  - area of memory for storage specific to function invocations
  - each function invocation: a new stack frame or activation record
  - same variable in different invocations stored in different stack frames: no conflict

**Stack Frames**
- Program stack pointed to by two registers
  - sp: stack pointer
  - fp: frame pointer
- New stack allocation at sp
- Stack accessed relative to fp
- Positive offsets: function arguments, link to frame of caller
- Negative offsets: local storage, e.g. \( b \Rightarrow fp - 8 \)

**Caller vs Callee**

**Arguments (std. Pentium)**
\[
gcd(x: \text{int}, y: \text{int}): \text{int} = \{ \ldots \} \]
- Arguments part of calling stack frame
- Pushed onto stack before return address (positive offset from fp)

**Local variables**
\[
gcd(x: \text{int}, y: \text{int}): \text{int} = \{
\text{if } (x == 0) \ y; \text{ else } \{
\text{if } (x < y) \{ t: \text{int} = x; x = y; y = t; \} \quad \text{gcd}(x\%y,y); \} \}
\]
- \( +12 \quad y \)
- \( +8 \quad x \)
- \( +4 \quad \text{ret pc} \)
- \( \text{fp} \Rightarrow +4 \quad \text{caller fp} \)
- \( \text{callee responsibility} \)
- \( \text{temp1} \)
- \( \text{sp} \Rightarrow -4 \quad \text{callee fp} \)
- \( \text{temp1} \)
- \( \text{t always at [fp-4]} \)

**Making a call**
\[
gcd(temp1,y)
\]
- **Caller:**
  - push \( y \)
  - push \( temp1 \)
  - call function
  - push ret addr
  - \( pc := gcd \)
  - \( sp := sp + 8 \)
- **On return:**
  - \( sp := sp - 8 \)
  - \( \text{temp1} \)
  - \( \text{ret addr} \)
  - \( \text{entries for call} \)
**Entering & leaving a function**

- **Callee**: need to establish new frame
- Push fp from calling frame
- Move sp to fp
- Adjust sp to make room for local variables
- On return:
  - move fp to sp
  - pop fp
  - return

**Modern architectures**

- Pentium calling conventions (for C): lots of memory traffic
- Modern processors: use of memory is much slower than register accesses
- Pentium has impoverished register set (6 somewhat general purpose registers)
- More registers ⇒ better calling conventions?

**MIPS, Alpha calling conventions**

- 32 registers! (actually 31: r0=0)
- Up to 4 arguments (6 on Alpha) passed in registers
- Return address placed in register (r31)
- No frame pointer unless needed
- Local variables, temporary values placed in registers

**MIPS stack frame**

- Caller: use jal gcd, r31
- leaf procedure:
  - Return addr in r31, sp = r30
  - K = max size of locals, temps
  - On entry: sp := sp - K
  - On exit: sp := sp + K; ret r31
  - fp := sp + K
- non-leaf procedure:
  - Put return addr on stack
  - Save temporary registers on stack when making call
  - On entry: sp := sp - K;
  - [sp + K - 4] := r31
  - On exit: r31 := [sp + K - 4];
  - sp := sp + K; ret r31;

**Mapping variables**

- Variables, temporaries assigned to locations during intermediate code generation (& optimization)
  - assigned to one of infinite number of registers initially
- Unoptimized code:
  - all registers mapped to stack locations
  - arguments pushed onto stack

**Compiling Functions**

- IR: intermediate code representation
- Stack frames store state for function calls
- Calling conventions
  - Pentium: everything on stack
  - MIPS, Alpha: everything in registers
- Next: code transformations for intermediate code generation