CS412/413

Introduction to
Compilers and Translators
Cornell University
Spring '00

Lecture 11: Stack frames

Administration

- HW2 is graded
- Programming Assignment 2 due Monday
- Homework 3 due Friday Feb. 25
- Prelim 1 on Wednesday, March 1

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: int ⇒ {...x : int...}
- Functions bound to function types

  gcd(x: int, y:int): int ⇒ {...gcd: int × int → int...}

Auxiliary environment info

- Entries representing functions are *not* normal environment entries

  { gcd: int ≥ int → 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

Handling Recursion

f(x: int): int = g(x) + 1   g(x: int): int = f(x) - 1

- Need environment containing at least

  {f: int → int, g: int → int}

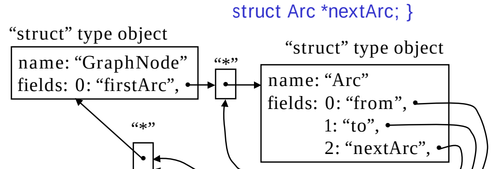
when checking both f and g
- Two-pass approach:
  - Scan top level of AST picking up all function signatures and creating an environment binding all global identifiers
  - Type-check each function individually using this global environment

Recursive Types

- Type declarations may be recursive too

Java: class List { Object head; List tail; }

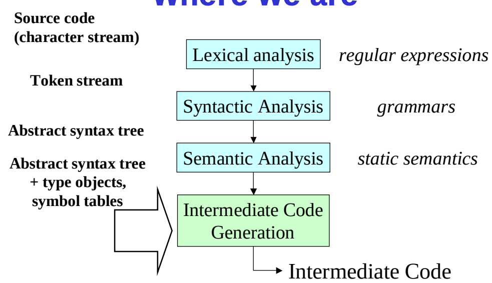
C: struct GraphNode { struct Arc *firstArc; }
struct Arc { struct GraphNode *from, *to;
struct Arc *nextArc; }



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

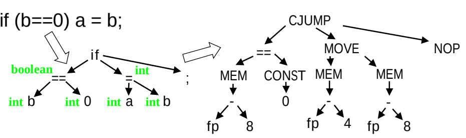
Where we are



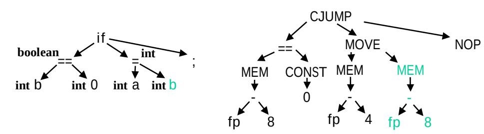
Our Intermediate Code

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

if (b==0) a = b;



Variables in IR



- Variables mapped to memory locations

  b ⇒ 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);
}
```

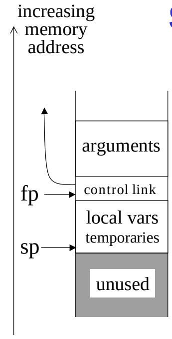
where to store x?

FORTRAN: stored in fixed memory locn!

Stack

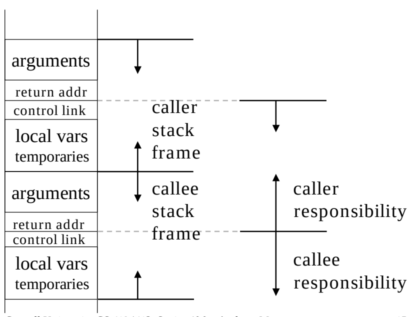
- Local storage allocated on *stack*
  - area of memory for storage specific to function invocations
  - each function invocation: a new *stack frame* or *activation record*
  - same variable in different invocations stored in different stack frames: no conflict

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 ⇒ fp - 8

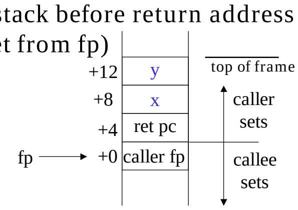
Caller vs Callee



Arguments (std. Pentium)

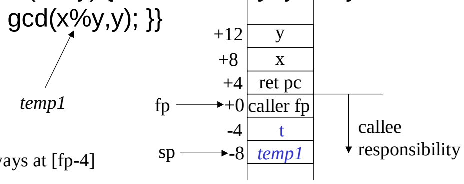
gcd(x:int, y:int): int = { … }

- Arguments part of calling stack frame
- Pushed onto stack before return address (positive offset from fp)



Local variables

```
gcd(x:int, y:int): int = {
  if (x == 0) y; else {
    if (x < y) { t:int = x; x = y; y = t; }
    gcd(x%y,y); }}
```



t always at [fp-4]

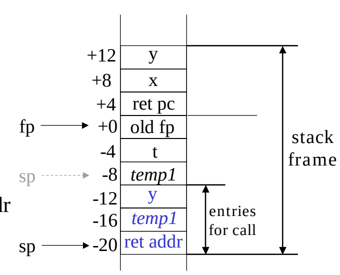
Making a call

gcd(*temp1*,y)

- **Caller**:
  - push y
  - push *temp1*
  - call function
    - push ret addr
    - pc := gcd

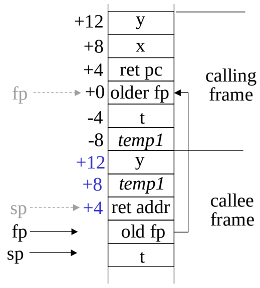
On return:

sp := sp + 8



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



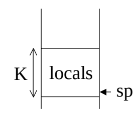
Modern architectures

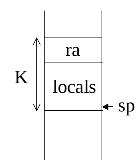
- Pentium calling conventions (for C): lots of memory traffic
- Modern processors: use of memory is much slower than register accesses
- Pentium has impoverished register set (6 somewhat general purpose registers)
- More registers ⇒ better calling conventions?

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 += 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