Administration

• Assignment 2
  – Do assignment 1 again, with visitors this time
  – Due tomorrow

• Project, part 1
  – Extend a compiler for the Expressions language to compile the Functions language instead
  – Work in groups of 2
Type checking

• Done bottom-up
  – The type of an expression is determined completely by the types of its sub-expressions
  • Some languages break this rule – ignore them!
Type checking II

• Based on a collection of type rules
  – E1 + E2 is correct and has type int if E1 and E2 are both correct and both have type int
  – I = E is correct if ...
  – E1 ? E2 : E3 is correct and has type ... if ...
  – some operators are polymorphic
    • work on operands with more than one type and can return results with more than one type

• The challenge is often making sure you haven’t forgotten any rules
Type checking - Complications

- **Subtypes**
  - Subclasses: class A extends B
  - Interfaces: class C implements D
  - Subranges: char, short, int, long, long long

- **Type conversions**
  - Casts
  - Widening vs. narrowing

- **Implicit type conversions**
  - double x = 3;
  - double x; int i; x + i
Activation records

• The low-level languages implemented by processors (e.g., x86, MIPS, PowerPC, ...) do not directly support procedures.

• We must translate procedures such as the one below into the instructions supported by a typical processor.

```c
int fac(int n) {
    if (n==0)
        return 1;
    else {
        return n*fac(n-1);
    }
}
```
Activation records

• First let’s examine this example, but before we do that let’s change it a little... low level languages do not support complex expressions

```c
int fac(int n) {
    int tmp1, tmp2;
    if (n==0)
        return 1;
    else {
        tmp1 = n-1;
        tmp2 = fac(tmp1);
        return n*tmp2;
    }
}
```
Activation records

• Questions that we have to answer:

```c
int fac(int n) {
    int tmp1, tmp2;
    if (n==0)
        return 1;
    else {
        tmp1 = n-1;
        tmp2 = fac(tmp1);
        return n*tmp2;
    }
}
```

• Where do we keep local variables like tmp1 and tmp2?

• How does a function receive arguments and return results?

• How does a function know where to return to? Where is the caller?
Activation Records & Stacks

• Many languages (e.g., C, Java, C++, Pascal, Modula, ...) allocate function/procedure activation records on a stack.

• This is the model we will study.

• Brief thought experiment:
  – What language features can be implemented with something simpler than a stack?
  – What language features require something “richer” than a stack?
Stack memory layout

- Traditionally, stacks start at high memory addresses and grow towards 0.
- Hardware instructions support this convention.
Stack & Heap

- A common way to divide memory between stack and heap.

Diagram:
- Heap: grows to the right
- Stack: grows to the left
- SP: points to the allocated memory
- Allocated memory
- Free memory
Actually ...

```
00400000-00402000 r-xp 00000000 00:34 12406175 /home/n/norm/stack/stack
00601000-00602000 r--p 00001000 00:34 12406175 /home/n/norm/stack/stack
00602000-00603000 rw-p 00002000 00:34 12406175 /home/n/norm/stack/stack
00603000-00629000 rw-p 00000000 00:00 0 [heap]
7f4632dc7000-7f4632f4e000 r-xp 00000000 08:01 917554 /lib64/libc-2.14.1.so
7f4632f4e000-7f463314d000 ---p 00187000 08:01 917554 /lib64/libc-2.14.1.so
7f463314d000-7f4633151000 r--p 00186000 08:01 917554 /lib64/libc-2.14.1.so
7f4633151000-7f4633152000 rw-p 0018a000 08:01 917554 /lib64/libc-2.14.1.so
7f4633152000-7f4633157000 rw-p 00000000 00:00 0
7f4633157000-7f4633177000 r-xp 00000000 08:01 917547 /lib64/ld-2.14.1.so
7f4633333000-7f4633336000 rw-p 00000000 00:00 0
7f4633375000-7f4633377000 rw-p 00000000 00:00 0
7f463377000-7f463378000 r--p 00020000 08:01 917547 /lib64/ld-2.14.1.so
7f463378000-7f463379000 rw-p 00021000 08:01 917547 /lib64/ld-2.14.1.so
7f463379000-7f463379a000 rw-p 00000000 00:00 0
7ffa9443000-7ffa9464000 rw-p 00000000 00:00 0 [stack]
7ffa955b000-7ffa955c000 r-xp 00000000 00:00 0 [vdso]
ffffffffffff600000-ffffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Introduction (chapter 1)
Stack Frame Layout

- Is essentially a design issue: decided by the language implementor.
- Depends on the programming language
- Depends on the architecture
- Sometimes the OS/Architecture dictates a certain frame structure.
- Language implementor may still choose to make their own design, but at the price of incompatibility (with libraries compiled by other compilers / languages).
- Let’s first think about what should be in a frame...
Stack Frame Layout

• Back to this function:

```c
int fac(int n) {
    int tmp1, tmp2;
    if (n==0)
        return 1;
    else {
        tmp1 = n-1;
        tmp2 = fac(tmp1);
        return n*tmp2;
    }
}
```

• What steps are involved in calling this function and returning from it?
Fixed or Variable Frame Size?

• Why might a frame be variable in size?
Registers

• We’ve already seen that there is a stack pointer (SP) register
  – On x86_64 this is %rsp

• We conventionally use a frame pointer (FP) register
  – On x86_64 this is %rbp

• What about other registers?
  – On x86_64 there are 14 more: %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15

• How are these registers shared between the caller and the callee of a function?
Registers

• What happens to the register if you call another function?
• Who saves a register?
  – caller save vs callee save.
  – architecture may specify the convention
• Which is better: caller save or callee save?
Parameter Passing

• Parameters may be passed in registers or in the stack frame.
• Caller and callee must agree exactly on how parameters get passed (and results get returned).
• Which is better: parameters in the frame or in registers?
• How does using registers for parameters interact with requirements to save registers?
Parameter Passing

• Does passing parameters in registers actually help (seeing that you might still need to save them on the stack anyway)?

• Yes, because:
  – Leaf procedures are the majority
  – Inter-procedural register allocation
  – No need to save if not used after the call.
  – Some architectures have register windows (although time has shown that this was a bad idea).
Return Addresses

- On x86_64 the CALL instruction pushes the return address on the stack => return address in the frame.
- Some processors use a dedicated register instead (because this is faster than memory access).
- Procedures are responsible for saving the register if they want to re-use it (i.e., if they call another proc).
Frame Resident Variables

• Local variables and temporary results (e.g., from complex expressions) can be kept
  – in registers
  – in the frame

• Which one is better?

• Is this always possible?
The big stack frame picture

- Locals for caller
  - variables
  - save locations for registers
- Parameters
- Return address (pushed by the machine)
- Saved frame pointer
- Locals for callee
  - variables
  - save locations for registers
- ...

Introduction (chapter 1)
X86_64 Register usage

• Parameter registers:
  – %rdi, %rsi, %rdx, %rcx, %r8, %r9

• Caller save registers:
  – %rax, %r10, %r11

• Callee save registers:
  – %rbx, %r12, %r13, %r14, %r15

• Special registers:
  – %rsp, %rbp
Parameter / return value complications

• Floating point values
  – parameters: xmm0-xmm7
  – return value: xmm0

• Structures
  – Parameters: if they fit in 2 registers, then in registers else on the stack
  – Result: if it fits in 2 registers then in %rax, %rdx, else on the stack

• Lots of parameters
  – On the stack

• How do you figure this out? Experiment! test.c