Administration

• Functions Project, phase 2
  – Extend the “static semantics” of Expressions to Functions
  – Identifiers and scopes
  – Types
    ```c
    int fact(int n) {
        nminus1 = n - 1;
        return n < 1 ? 1 : n * fact(nminus1);
    }
    ```
    ```c
    print fact(10)
    ```
  – Deadline is next Wednesday
OK, back to translating to IR

- Now that we have seen
  - the IR tree model that we are using and...
  - the “scaffolding” we use to generate code for different types of expression contexts
- Let’s look at a series of specific language constructs and think about how to generate IR code for them.
Translation to IR

- Relatively straightforward but many cases. Let’s look at a few:
  - 3
  - 3 + 4
  - x < 1
    - l (a local variable in a function)
    - p (a parameter to a function)
    - p = p + 1
    - f(3)
    - int f(int x) { … }
    - condition ? expression1 : expression2
Simple (Local) Variables

- Example:

```c
int f(int p) {
    l = 4;
    return l + p
}
```
Simple (Local) Variables

• Example:

```c
int f(int p) {
    l = 4;
    return l + p
}
```

• Allocate a temp for each local variable
Simple Parameters

• Example:

```c
int f(int p) {
    l = 4;
    return l + p
}
```
Simple Parameters

• Example:

```c
int f(int p) {
    l = 4;
    return l + p
}
```

• The machine architecture tells us where they are.
• Different architectures put them in different places
• We want our translation to be (as much as possible) architecture neutral
The Frame Class

• Frame class has abstract methods to
  – create an object to keep track of the current frame's layout (factory methods)
  – allocate variables / formals in the frame
  – special temporaries (the FP and RV)
  – other target architecture specific stuff

• Designed to hide details such as frame layout and whether locals/formals are in registers or in the frame
The Frame Class

Frame

Abstract class

X86_64Frame

PPCFrame

MIPSFrame
The Frame Class

```java
public abstract class Frame {
    public abstract Frame newFrame(Label name, int nFormals);

    /** A label that points to the beginning of the function's code. */
    public abstract Label getLabel();

    /** fetch a list of abstract representations of the “addresses” of
     * the formal parameters. */
    public abstract List<Access> getFormals();

    /** Allocate space for a local variable in this frame. */
    public abstract Access allocLocal();

    /** Frame pointer (e.g. a temp mapped to %rbp on x86_64) */
    public abstract Temp FP();

    /** Return value (e.g. a temp mapped to %rax on x86_64) */
    public abstract Temp RV();
    ...
}
```
/**
 * An instance of this class represents a place to store a local, \texttt{temp}
 * or parameter.
 *
 * It may be a register or memory address relative to the current
 * stack frame.
 *
 * This class is abstract for two reasons.
 *
 * First, there are concrete subclasses for the different cases
 * (at least two: one for register and one for \texttt{inFrame}).
 *
 * Second, each architecture can provide its own concrete
 * implementations.
 */

\textbf{public abstract class} Access {

/**
 * Translate into intermediate representation (returns an \texttt{IRExp}
 * that can be used either as an L-value or an R-value
 */

\textbf{public abstract} IRExp exp(Temp fp);
}
X86_64Frame Implementation

• Formals:
  – allocated as “InRegister” for the first 6, “InFrame” for the rest

• Locals:
  – allocated as InRegister

• Running out of Registers (i.e., Temps)?
  – don't worry now, this phase assumes an infinite # of Temps
  – worry later: “Register Allocation” phase
OK, back$^2$ to translating to IR

• Now that we have seen
  – the IR tree model that we are using and...
  – the “scaffolding” we use to generate code for different types of expression contexts...
  – the Frame infrastructure for accessing parameters and locals

• Let's look at a few more specific language constructs and think about how to generate IR code for them.
Translation to IR

• Relatively straightforward but many cases. Let’s look at a few:

- 3
- 3 + 4
- x < 1
- l (a local variable in a function)
- p (a parameter to a function)
- p = p + 1
- f(3)
- int f(int x) { … }
- condition ? expression1 : expression2
Simple Parameters

- Example:

```c
int f(int p) {
    x = 4;
    return x + p
}
```

- Given the Frame class, what do we generate for p?
Translation to IR

- Relatively straightforward but many cases. Let’s look at a few:
  - 3
  - 3 + 4
  - x < 1
  - l (a local variable in a function)
  - p (a parameter to a function)
    - p = p + 1
    - f(3)
    - int f(int x) { … }
    - condition ? expression1 : expression2
Assignment statements

• These are easy...
• We have an IR code to represent assignment.

\[ p = p + 1 \]

\[ \Rightarrow \text{MOVE( xlate(p), xlate(p + 1) )} \]
Translation to IR

• Relatively straightforward but many cases. Let’s look at a few:

  ✓  – 3
  ✓  – 3 + 4
  ✓  – x < 1
  ✓  – 1 (a local variable in a function)
  ✓  – p (a parameter to a function)
  ✓  – p = p + 1
    – f(3)
    – int f(int x) { … }
  – condition ? expression1 : expression2
Function Calls

- These are easy...
- We have an IR code to represent function calls directly.
- \( f(<a>,<b>,<c>) \)
- \( \Rightarrow \text{CALL( NAME(f\_label),list(<a>,<b>,<c>))} \)
Translation to IR

- Relatively straightforward but many cases. Let’s look at a few:

  - 3
  - 3 + 4
  - x < 1
  - 1 (a local variable in a function)
  - p (a parameter to a function)
  - p = p + 1
  - f(3)
    - int f(int x) { ... }
    - condition ? expression1 : expression2
Function Declarations

• The body is compiled into IR
  – creating a “ProcedureFragment” that is stored in a global list of fragments.

• We’ll have to add some additional “glue” code
  – linking activation records
  – allocating space

```java
int foo(int p1, boolean p2) {
    x = p1 * 2;
    y = p2 ? x : x * 2;
    return y;
}
```
Translation to IR

- Relatively straightforward but many cases. Let’s look at a few:
  - 3
  - 3 + 4
  - x < 1
  - l (a local variable in a function)
  - p (a parameter to a function)
  - p = p + 1
  - f(3)
  - int f(int x) { … }
  - condition ? expression1 : expression2
We want ?: to be non-strict (do you remember what that means from 311?)

Why?

How?

```c
int fac (int n) {
    return n < 1 ? 0 : n * fac(n - 1);
}
```
What’s left?

- if (c) { s1 } else { s2 }
- while (c) { s }
- do { s } while (c)
- for (init ; c ; inc) { s }