Administration

• Functions Project, phase 1
  – Extend the parser for the Expressions language to parse the Functions language instead

```c
int fact(int n) {
    nminus1 = n - 1;
    return n < 1 ? 1 : n * fact(nminus1);
}

print fact(10)
  – Deadline is tomorrow
```
Translation to Intermediate Code (Chapter 7)

• This stage converts an AST into an intermediate representation.
• Last stage of “semantic analysis”: the IR can be thought of as a representation of the “meaning” of the program in more low-level terms (without committing to a specific target machine).
Why use an Intermediate Representation (IR)?

- More portable compiler architecture

```
Java
ML
Pascal
C
C++
```
```
Arm
Sparc
MIPS
PowerPC
x86
x86_64
```
Why use an Intermediate Representation (IR)?

- Better separation of concerns:
  - Frontend not complicated with details of target architecture
  - Backend not complicated with details of the language semantics
Design of a good IR

• What do we seek from a good IR?
  – convenient to produce by semantic analysis phase
  – convenient to translate to machine language (for all desired architectures)
  – clear and simple meaning so that optimizing transformations can be specified and implemented easily.
IR Trees

Exp ::= CONST( int )
    | NAME( Label )
    | TEMP( Temp )
    | BINOP( Op, Exp, Exp)
    | MEM(Exp)
    | CALL( Exp fun, List<Exp> args)
    | ESEQ( Stm, Exp )
Stm ::= MOVE(Exp dst, Exp src)
    | EXP(Exp)
    | JUMP(Label)
    | CJUMP(RelOp, Exp, Exp, Label thn, Label els)
    | SEQ( Stm, Stm)
    | LABEL( Label )
Data ::= DATA(Label, List<Exp> values)
Translation to IR

- Relatively straightforward but many cases. Let’s look at a few:
  - 3
  - $3 + 4$
  - $p$ (a parameter to a function)
  - $l$ (a local variable in a function)
  - $f(3)$
  - $p = p + 1$
Translation to IR

• To emit good code we may need to generate different code for an expression / statement based on its usage context.

• For example, one expression:
  – $x < 1$

• Two contexts
  – $z = x < 1$
  – $z = x < 1 ? 1 : \text{fac}(x - 1)$

• Trouble is a combinatorial explosion
  – expressions
  – usage contexts
“Context Dependent” Translation

• To generate “nice” code, we need to be able to generate different code for some expressions depending on how they are used

• We distinguish three usage contexts for an exp E
  – contexts that need an explicit value:
    \[ x = E \quad f(E) \quad \text{return } E \]
  – contexts that discard the expression value:
    \[ E; \]
  – contexts that use E’s value as a branching condition.
    \[ E ? ... : ... \]
    \[ \text{if } (E) \{ \ldots \} \text{ else } \{ \ldots \} \]
“Context Dependent” Translation

• Instead of returning an actual IRExp (or IRStm) our translator returns an object that knows how to create good code for these three different kinds of contexts:

```java
public abstract class TRExp {
    /**
     * An explicit result value is required.
     */
    abstract IRExp unEx();

    /**
     * The value is discarded (good compilation strategy may avoid producing it).
     */
    abstract IRStm unNx();

    /**
     * The value is used as the condition for a CJUMP.
     */
    abstract IRStm unCx(Label ifTrue, Label ifFalse);
}
```
Example: Context dependent Assign Exp

```java
public TRExp visit(Assign n) {
    final Access var = currentEnv.lookup(n.name);
    TRExp val = n.value.accept(this);
    final IRStm stm = IR.MOVE(
        var.exp(frame.FP()), val.unEx());

    return new TRExp() {
        IRExp unEx() {
            return ESEQ(stm, var.exp(frame.FP()));
        }
        IRStm unNx() {
            return stm;
        }
        IRStm unCx(Label ifTrue, Label ifFalse) {
           ...
        }
    };
}
```
The Ex class

- Many AST nodes naturally translate to expressions that produce an explicit value.
  - To use them as a statement we must discard that value.
  - To use them as a branch condition we must generate an actual branch with a test of the expression's value.
The Ex class

```java
public class Ex extends TRExp {
    private final IRExp exp;
    public Ex(IRExp exp) { this.exp = exp; }

    @Override IRStm unCx(Label t, Label f) {
        return CJUMP(RelOp.NE, exp, CONST(0), t, f);
    }

    @Override IRExp unEx() {
        return exp;
    }

    @Override IRStm unNx() {
        return EXP(exp);
    }
}
```
The Nx class

- Many AST nodes naturally translate to an IRStmt that produces no value.
  - It should be illegal to use them in a context that wants a value (including as a branch condition).
  - If your type checker works, then the unCx and unEx cases are illegal and needn’t have a real implementation.
public class Nx extends TRExp {

    private final IRStm stm;

    public Nx(IRStm stm) { this.stm = stm; }

    public IRStm unCx(Label ifTrue, Label ifFalse) {
        throw new Error("Bug in type checker?");
    }

    public IRExp unEx() {
        throw new Error("Bug in type checker?");
    }

    public IRStm unNx() {
        return stm;
    }
}
The Cx class

• Some AST expressions most naturally translate to a CJUMP (e.g., comparing numbers with < can most naturally be done with a CJUMP!)

• We can convert them into an expression that returns an explicit value by using that branch to store an explicit value (typically 0 for false, and 1 for true).
The Cx class

```java
public abstract class Cx extends TRExp {
    abstract IRStm unCx(Label ifTrue, Label ifFalse);
    IRExp unEx() {
        Label t = new Label(), f = new Label();
        TEMP r = TEMP(new Temp());
        return ESEQ( SEQ( MOVE(r, TRUE),
                          unCx(t,f),
                          LABEL(f),
                          MOVE(r, FALSE),
                          LABEL(t)),
                     r);
    }
    IRStm unNx() {
        Label end = new Label();
        return SEQ( unCx(end, end),
                    LABEL(end));
    }
}
```
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.
Simple (Local) Variables

• Example:

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

• Allocate a temp for each local variable
Simple Parameters

• Example:

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

Abstract class

Frame

X86_64Frame

PPCFrame

MIPSFrame
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();
    ...
}
The Access Class

/**
 * An instance of this class represents a place to store a local, 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 inFrame).
 *
 * Second, each architecture can provide its own concrete
 * implementations.
 */

public abstract class Access {
  /**
   * Translate into intermediate representation (returns an IRExp
   * that can be used either as an L-value or an R-value
   */

  public abstract IRExp exp(Temp fp);
}

Introduction (chapter 1)
X86_64Frame Implementation

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

• Locals:
  – allocated as InReg

• 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\textsuperscript{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.
Simple Parameters

• Example:

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

• Given the Frame class, what do we generate for p?
Function Calls

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

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