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

Diagram:

- Java
- ML
- Pascal
- C
- C++
- IR
- 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 )
    | CMOVE( RelOp, Exp, Exp, Exp dst, Exp src )
    | 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
  – \( x < 1 \)
  – \( p \) (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 : 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  f(E)  return E
    ```
  - contexts that discard the expression value:
    
    ```
    E;
    ```
  - contexts that use $E$’s value as a branching condition.
    
    ```
    E ? ... : ...
    if (E) { ... } else { ... }
    ```
“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);
}
```
Assignment statements as expressions

• In C and Java, assignment statements aren’t statements, they are expressions:
  – i = j = 10;
  – while ((nread = in.read(buf, 4096)) > 0) {
    ...
  }
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.
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.
The Nx class

```java
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.
Translation to IR

• Relatively straightforward but many cases. Let’s look at a few:
  – 3
  – 3 + 4
  – x < 1
  – p (a parameter to a function)
  – l (a local variable in a function)
  – f(3)
  – p = p + 1
Example: <

```java
TRExp visit(LessThan n) {
    TRExp l = n.e1.accept(this);
    TRExp r = n.e2.accept(this);
}
```