This Lecture:
- Background: A long-term, work-in-progress, personal exploration: how do I express myself in the code that I write?
- My goal today: share some thoughts of mine, provoke some thoughts in you
- Themes:
  - Tangling and crosscutting concerns
  - Meta-programming/program generation
  - Literate programming

Context:
- This is a talk about programming (no theory)
- Different programming languages teach us:
  - to think in different ways
  - to develop problem solving skills
- But any single language ultimately constrains the way that a programmer thinks ...
- The "Tyranny of the Dominant Decomposition" (Ossher & Tarr)
Hierarchical Modularity Mechanisms:
- programs
- packages
- classes
- methods
- statements
- expressions

Tangling Considered Inevitable:
In the implementation of any complex system ...
... when programs are written using the hierarchical modularity mechanisms of current programming languages ...
... and as they evolve to add new functionality ...
... tangling is inevitable

Crosscutting Concerns:
There will be concerns that one would like to modularize, but for which the code will be spread out ...

Tangling Considered Harmful:
When programmers are forced to weave different aspects of functionality together, the end results are likely to be:
- Harder to write
- Harder to maintain
- Less reliable
- Harder to reuse
- Less portable

What if our tools did this for us instead?

Equalities on Program Trees:
If our programming language admitted structure changing equalities ...

Modularizing Change:
- programs
- packages
- classes
- methods
- statements
- expressions
... then perhaps we could factor out new code more easily as a separate structure ...
Same Program, Different Views:

- Multiple, equivalent views of a program’s source code
- Equivalences described by localized tree transformations
- Potential for additional meta-programming functionality at nodes

Hacks Ahead!

You are now entering a theory-free zone

Experiences in Compiler Construction

The mini Java compiler, mjc:

- mjc is a compiler for a subset of Java that produces IA32 assembly code for the 386 and later processors
- mjc was written to accompany my compiler class

The Structure of CS32x:

- Compilers are often structured as a pipeline of separate phases ...

  ![Diagram of compiler phases]

- It makes sense to structure a compiler class in a similar way ...

- But some parts of the code for mjc don’t fit this pattern!

Representing Compiler Phases:

- Compiler phases represented by an abstract type Phase:

  ![Diagram of Phase class]

- Different Phases serve very different purposes, but they all have the potential to generate diagnostics.
- Different Sources obtain their input in different ways, but each one can be called on to return a sequence of input lines.
- Some components are specific to Mini Java, others are potentially language independent
Using Inheritance (1):

- In examples like these, we use inheritance to document the existence of multiple implementations of a particular interface ...
- Can introduce code for these phases incrementally
- Structure of code corresponds to structure of presentation

Putting the Pieces Together:

```java
Source source = new JavaSource(handler, descr, reader);
MjcLexer lexer = new MjcLexer(handler, source);
MjcParser parser = new MjcParser(handler, lexer);
ClassType[] classes = parser.getClasses();
```

Putting the Pieces Together:

```java
Source source = new JavaSource(handler, descr, reader);
MjcLexer lexer = new MjcLexer(handler, source);
MjcParser parser = new MjcParser(handler, lexer);
ClassType[] classes = parser.getClasses();
```

Factoring out Error Handling:

- It would be nice if I could factor out error handling as a separate aspect:
- I haven’t figured out how to do that yet ...

Representing Abstract Syntax:

- The tree structures that are used to representing abstract syntax are described by a collection of classes
- There are more than 60 of these classes, which can seem quite daunting
- Understanding how these classes are organized can make things much easier to follow

Using Inheritance (2):

- In examples like these, we use inheritance to encode datatypes ...
- Similar to `datatype/data` definitions in ML/Haskell
Abstract Syntax in mjc:

- Literal
- BooleanLiteral
- IntLiteral
- NullLiteral
- FieldAccess
- LeftHandSide
- BitOpExpr
- EqOpExpr
- LogicOpExpr
- RelOpExpr
- NumericOpExpr
- BinaryOpExpr
- Invocation

Syntax: Binary Operator Expressions:

- Expressions with binary operators have two operands:
  new BinaryOpExpr(pos, expr1, expr2)

Bits:
- BitAndExpr
- BitOrExpr
- BitXorExpr
- EqualExpr
- NotEqualExpr
- CondAndExpr
- CondOrExpr
- GreaterThanExpr
- LessThanExpr
- AddExpr
- MulExpr

Statements:

- mjc supports only a few kinds of statement, but these are enough to write interesting programs:

```java
{ stmt1 ... stmtn } ;
if expr then stmt1 [else stmt2]
return [expr];
while (expr) stmt
```

An Example:

```java
public class While extends Statement {
  private Expression test;
  private Statement body;
  public While(Position pos, Expression test, Statement body) {
    super(pos);
    this.test = test;
    this.body = body;
  }
  ... static analysis/code generation goes here!
}
```

... continued:

```java
public class While extends Statement {
  ...
  boolean check(Context ctxt, Env env) {...}
  void compile(Assembly a) {...}
  void compileThen(Assembly a, Label l) {...}
}
```

... continued

The `While` class encapsulates all of the features of a `while` statement in one place:

- **Perfect**, if you want to use it as a model for adding a `repeat until` construct
- **Not at all convenient**, if you want to understand later phases of the compiler (e.g., type checking):
  - Can't see parts corresponding to other constructs
  - Other, irrelevant features obscure your view
Back in CS32x ...

- I don’t attempt to talk about all the different classes any more ... a couple of examples will/have to suffice...
- The Java encoding mixes essential details with irritating noise ... harder for students to identify and focus on key parts, and greater potential for errors to creep in...
- We can’t package up “static analysis”, “type checking”, or “code generation” as modular chunks of code ... Instead, we must scatter little bits in each of the abstract syntax classes...
- diffs are a useful tool for quick patches, but not a good vehicle for reliable software composition.

Observations:

- There is a lot of boilerplate in declaring classes, attributes, and constructors:
  - uninspiring to code
  - easy to make mistakes
  - painful to change

The Sweet Approach

A List Datatype:

```java
public abstract class List {
  //List
  abstract public int length();
}
public class Nil extends List {
  //Nil
  public Nil() {
  }
  public int length() {
    return 0;
  }
}
public class Cons extends List {
  //Cons
  private int head;
  private List tail;
  public Cons(int head, List tail) {
    this.head = head;
    this.tail = tail;
  }
  public int length() {
    return 1 + tail.length();
  }
}
```

A simple enough idiom ... but tedious to write ...

A List Datatype:

```java
public abstract class List {
  //List
  abstract public int length();
}
public class Nil extends List {
  //Nil
  public Nil() {
  }
  public int length() {
    return 0;
  }
}
public class Cons extends List {
  //Cons
  private int head;
  private List tail;
  public Cons(int head, List tail) {
    this.head = head;
    this.tail = tail;
  }
  public int length() {
    return 1 + tail.length();
  }
}
```

A simple enough idiom ... but tedious to write ...

A List Datatype:

```java
public abstract class List {
  //List
  abstract public int length();
}
public class Nil extends List {
  //Nil
  public Nil() {
  }
  public int length() {
    return 0;
  }
}
public class Cons extends List {
  //Cons
  private int head;
  private List tail;
  public Cons(int head, List tail) {
    this.head = head;
    this.tail = tail;
  }
  public int length() {
    return 1 + tail.length();
  }
}
```

Duplication of information!
A List Datatype:

```java
public abstract class List {
    abstract public int length();
}

public class Nil extends List {
    public Nil() {}
    public int length() { return 0; }
}

public class Cons extends List {
    private int head;
    private List tail;
    public Cons(int head, List tail) {
        this.head = head;
        this.tail = tail;
    }
    public int length() {
        return 1 + tail.length();
    }
}
```

Tangling of datatype definitions with the implementations of operations.

First Attempt:

- Capture essential details in a data structure, and use a (Haskell) program to generate all the boilerplate:
  ```haskell
  [Node "List" [] ["public int length()"],
   Node "Nil" [] [],
   Node "Cons" ["int", "head"], ["List", "tail"] [] []]
  ```
- Nice idea ...but:
  - Deeply nested expressions like this are not easy to write (or get right) in Haskell syntax
  - The data structures become increasingly complicated as new features are added ...
  - Once you start modifying the generated code, it's hard to change the datatype.

Generating Skeletons is Not Enough!

- We generate code and then modify it, by hand, to add the implementations of any operations.
- These modifications are not captured in the real source code for the program.
- We need a new static weaving and editing tool:

Introducing Sweet:

```java
public abstract class List {
    public case Nil
    public case Cons(private int head, 
                     private List tail)
}

public int length()
    case List abstract;
    case Nil { return 0; }
    case Cons { return 1 + tail.length(); }
```

Down with boilerplate!
Down with repetition!
Down with tangling!

Hmm, this looks familiar!

```haskell
data List = Nil 
          | Cons { head :: Int, tail:: List }

length :: List -> Int
length Nil = 0
length (Cons {head,tail}) = 1 + length tail
```

But looks can be deceiving ...

Pizza Recipes, not Pizzas:

```java
public abstract class List {
    public case Nil
    public case Cons(private int head, 
                     private List tail)
}

public int length()
    case List abstract;
    case Nil { return 0; }
    case Cons { return 1 + tail.length(); }
```

Poor man's meta-programming: the input to sweet is not a program ... it's a description of how to construct/extend a program ...

```haskell
data List
    = Nil 
    | Cons { head :: Int, tail:: List }

length :: List -> Int
length Nil = 0
length (Cons {head,tail}) = 1 + length tail
```
Extensibility:
To build a bigger program, add instructions to define new addresses, and new content:

```java
public class Append(private List l, private List r) extends List {
    public int length() {
        return l.length() + r.length();
    }
}
```

You can’t do that in Haskell!

```haskell
data List = Nil |
  Cons { head :: Int, tail:: List } |
  Append { l :: List, r :: List }

length :: List -> Int
length Nil = 0
length (Cons head tail) = 1 + length tail
length (Append l r) = length l + length r
```

Visitors in Java (infrastructure):

```java
interface ListVisitor {
    int visitNil(Nil nil);
    int visitCons(Cons cons);
}
class List { ... abstract int accept(ListVisitor v); }
class Nil { ...
    int accept(ListVisitor visitor) {
        visitor.visitNil(this);
    }
}
class Cons { ...
    int accept(ListVisitor visitor) {
        visitor.visitCons(this);
    }
}
```

A lot of boilerplate
Can't be generated within running Java code
Adding a new Append class would still be a crosscutting concern

Visitors in Java (instantiation):

```java
class LengthVisitor implements ListVisitor {
    int visitNil(Nil nil) { return 0; }
    int visitCons(cons cons) {
        return 1 + cons.getTail().accept(this);
    }
}
class SumVisitor implements ListVisitor {
    int visitNil(Nil nil) { return 0; }
    int visitCons(cons cons) {
        return cons.getHead() + cons.getTail().accept(this);
    }
}
```

Requires us to break object encapsulation

Visitors in Java (... continued):

```java
class MemberVisitor implements ListVisitor {
    private int elem;
    MemberVisitor(int elem) {
        this.elem = elem;
    }
    boolean visitNil(Nil nil) { return false; }
    boolean visitCons(cons cons) {
        return cons.getHead()==elem || cons.getTail().accept(this);
    }
}
```

Static typing may require us to create a whole family of visitors!

The Program Grid:

```
<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>f(C)</td>
<td>f(D)</td>
<td>f(E)</td>
</tr>
<tr>
<td>g</td>
<td>g(C)</td>
<td>g(D)</td>
<td>g(E)</td>
</tr>
<tr>
<td>h</td>
<td>h(C)</td>
<td>h(D)</td>
<td>h(E)</td>
</tr>
</tbody>
</table>
```
Same Program, Different Views:

```
<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>f(C)</td>
<td>f(D)</td>
<td>f(E)</td>
</tr>
<tr>
<td>g</td>
<td>g(C)</td>
<td>g(D)</td>
<td>g(E)</td>
</tr>
<tr>
<td>h</td>
<td>h(C)</td>
<td>h(D)</td>
<td>h(E)</td>
</tr>
</tbody>
</table>
```

Breaking the "Tyranny of the Dominant Decomposition"

Back to mjc:
- In preliminary experiments, I have used sweet to refactor the code for mjc, separating out different aspects for:
  - Abstract syntax
  - Static analysis
  - Typing
  - Code generation
  - Etc...
- And I have also found bugs in mjc resulting from duplication in the original code ...
- An ongoing project, yet to be inflicted on my compilers students ...

Case Study: Building a Lexical Analyzer

Building a Lexical Analyzer:
- Lexical analysis is a process that breaks an input stream into a series of "tokens" and returns a code to describe each one.
- How do we add a new token to a lexer?
  1. Select a new code to identify the token
  2. Map the token string to the token code
  3. Update code to recognize new token code
Step 1: Define token code

```java
public class Alphabet {
    ...
    public static final int WHILE = 63;
    public static final int DO     = 64;
    ...
}
```

Be sure to pick a distinct code for each token!
But the specific choice doesn't matter ...

Step 2: Map string to code

```java
public class Alphabet {
    Alphabet() {
        ...
        addReserved("while", WHILE);
        addReserved("do",     DO);
        ...
    }
}
```

private Hashtable reserved = new Hashtable();
private void addReserved(String str, int token) {
    reserved.put(str, new Integer(token));
}

Step 3: Recognize new code

```java
public class Alphabet {
    ...
    public String describeToken(int token) {
        switch (token) {
            case WHILE : return "while keyword";
            case DO     : return "do keyword";
            ...
            default     : return "unknown token";
        }
    }
}
```

default is required in case we forget to cover one or more of the token codes ...

Using Sweet Syntax:

```java
macro AddReservedSymbol(TOK, lexeme, description) {
    class Alphabet {
        public static final int TOK = @fresh;
        Alphabet() { addReserved(lexeme, TOK); }
        @describeTokens { case TOK : return description; }
    }
}
```

```
macro AddReservedSymbol(WHILE, "while", "while keyword")
```

All three steps combined in one definition ...

```
macro AddReservedSymbol(WHILE, "while", "while keyword")
```

All three steps invoked in one call ...

Sweet Notation:

- @fresh generates a unique integer code for every source code occurrence
- Add code to a particular class:
  - class Foo { ... new code goes here ... }
- Change code at a particular location:
  - address > { add to end }
  - address ! { replace }
  - address < { add at beginning }
  - Address can be a method or constructor name, or a source position, @Position

Code Positions, @name:

```java
public String describeToken(int token, String lexeme) {
    switch (token) {
        case ENDINPUT : return "end of input";
        case INTLIT   : return "int literal, " + lexeme;
        case STRINGLIT: return "string literal, " + lexeme;
        case POPEN    : return "open parenthesis, \"\"\";"
        case PCLOSE   : return "close parenthesis, \"\"\";"
        ...
        default       : return "lexeme, \"\" + lexeme + \"\";"
    }
    @describeTokens
    return "lexeme, \"\" + lexeme + \"\";"
}
```
**Sweet Macros:**

- Define a new macro:
  - `macro M(arg1,...,argn) { ... code template here ... }`
  - Arguments `arg1, ..., argn` used in template

- Invoke a macro:
  - `macro M(val1, ..., valn)`
  - Expands to template for macro replacing each `arg` identifier with the corresponding `val` value

**Identifier Splicing:**

Multiple identifiers can be spliced together using a backslash:

```java
public class Xs(public X head, public X\s next)
}
macro AddCons(X) {
    class X {
        public X\s cons(X\s next) {
            return new X\s(this, next);
        }
    }
}
```

**Setters and Getters:**

Sweet provides some limited methods for deriving code automatically:

```java
public int setter getter foo;
```

This can almost be captured with macros:

```java
macro setter(T,X) { void set\X(T X) { this.X = X; } }
macro getter(T,X) { T get\X() { return this.X; } }
```

(But this is ugly; it duplicates type information, and doesn’t camel case the setter/getter names)

**Band-aid for a Bad Language?**

- The features that I’ve described are arguably just a work-around for:
  - missing features in the Java programming language
  - my unwillingness to use “standard” Java idioms

- But it’s not clear how to achieve some of these features using current language technology, especially if we care about static typing

- Some may one day become part of new language designs, albeit in a more general and elegant form

- Some will never be more than an ugly hack :-)

**Case Study: Regular Expressions**

A tutorial paper, describing the algorithm for converting regular expressions into NFAs and NFAs into DFAs
Executable code for the algorithm for converting regular expressions into NFAs and NFAs into DFAs

Abstract Syntax

We can generate all of the classes for representing the syntax of regular expressions from the following lines of sweet code:

```java
abstract class RegExp {
    case Epsilon
    case Char (private int c)
    case Seq (private RegExp r1, private RegExp r2)
    case Alt (private RegExp r1, private RegExp r2)
    case Rep (private RegExp r)
}
```

Functions on RegExps:

We can defined functions on RegExp values by a collection of cases:

```java
public String fullParens() {
    case RegExp abstract;
    case Epsilon { return "\"\"; }
    case Char { return Character.toString((char)c); }
    case Seq { return "( \" + r1.fullParens() + r2.fullParens() + \""); }
    case Alt { return "( \" + r1.fullParens() + "\"|\" + r2.fullParens() + \""); }
    case Rep { return "( \" + r.fullParens() + \"\")\"; }
}
```

RegExp Visitors:

We could even use sweet to automate the construction of some of the boilerplate for type-specific visitors:

```java
macro RegExpVisitor(T) {
    interface RegExpVisitor{T {
        T visitEpsilon{T(Epsilon epsilon)};
        ...
        T visitRep{T(Rep rep)};
    }
    T accept(RegExpVisitor{T visitor}) {
        case RegExp abstract;
        case Epsilon { return visitor.visitEpsilon{T(this)};
        ...
        case Rep { return visitor.visitRep{T(this)};
        ...
    }
```

Incremental Construction:

Define key parts of the representation for DFA/NFA state:

```java
class State {
    Transition[] trans = null;
    boolean accept = false;
}
```

Add more code later, to suit the narrative:

```java
class State {
    /** Output a description of this machine state. */
    void display() {
    ...
    }
}
```

Building NFAs:

```java
State tonFA(State s) {
    case RegExp abstract;
    case Epsilon { return s; }
    case Seq {
        return r1.tonFA(r2.tonFA(s));
    }
    case Alt {
        State n = new State();
        n.trans = new Transition[] {
            new Transition(r1.tonFA(n), new Transition(s));
        };
        return n;
    }
```

Read the full document at: http://www.cs.pdx.edu/~mpj/regexp.pdf
Case Study: Dependency Analysis

Problem: Given a directed graph ...

Dependency Analysis (SCCs):
- Represent/construct dependency graph
- Kosaraju and Sharir’s SCC algorithm: \( \text{dfs}_2(\text{dfs}_1(g)) \)
- First depth-first search \( \rightarrow \) ordered list of bindings
- Second depth-first search \( \rightarrow \) list of groups
- Pointer to binding group

Simple example/test
Further tests

Sweet Literate Programming:
- dep.sweet \rightarrow dep.tex
  - output.sweet \rightarrow test.sweet
  - test.sweet
  - further.sweet
- Binding.java \rightarrow Binding.class
  - bindingGroup.java
  - BindingGroup.java
  - Test.java
  - Main.java
- class Binding {
  - class Bindings {
  - Represent/construct dependency graph
  - Kosaraju and Sharir’s SCC algorithm: \( \text{dfs}_2(\text{dfs}_1(g)) \)
  - First depth-first search \( \rightarrow \) ordered list of bindings
  - Second depth-first search \( \rightarrow \) list of groups
  - Pointer to binding group
  - Simple example/test
  - Further tests
  
Sweet and Sour

Read the full document at: http://www.cs.pdx.edu/~mpj/dep.pdf
Sweet Problems: (1)

1) Code out of context:

```java
/** Check whether this statement is valid and return a boolean
 * indicating whether execution can continue at the next statement.
 */
public boolean check(Context ctxt, VarEnv env, int slot)
    case Statement abstract;
    case Block {
        return (stmts==null) || stmts.check(ctxt, env, slot);
    }
    case Empty {
        return true; // Always runs on ...
    }
    case ExprStmt {
        try {
            expr.checkExpr(ctxt, env);
        } catch (Diagnostic d) {
            ctxt.report(d);
        }
        return true;
    }
    -
```

where are these variables defined?

An IDE could show context as the user moves from one
program point to the next ...

Sweet Problems: (2 & 3)

2) Tracing back errors:
How do javac's errors relate back to sensible
diagnostics on sweet code?

Does sweet need to provide its own compiler?

How far can static checking go?

3) Incremental computation:
Minor changes in sweet code triggers complete rebuild!

Tyranny Transferred:

- With Sweet, the language no longer dictates
  a "dominant decomposition"

- But there is still a dominant decomposition,
  courtesy of the program's initial author

- When the regime is relaxed ... a new tyrant
  steps in!

Closing Thoughts

Work in Progress:

- There are known flaws in both the design and
  the implementation of Sweet

- Both design and implementation evolve (very
  slowly) as I work on case studies

- Sweet is a personal project, not published/
  reviewed research, not necessarily novel

- Writing literate code is time consuming, and
  limits agility

Questions to Ponder:

- How do you want to express yourself as an
  advanced programmer?

- What are your goals as an author?

- How do current languages and tools help or
  hinder you?

- How would you write programs if you had the
  freedom to choose your own decomposition?