Introduction

Tristan Naumann
Sahar Hasan
Steve Hehl
Brian Lu
Friendly Interactive Recursion Educator (FIRE)

- Recursion is one of the most difficult topics for novice programmers to understand
- Teach through interactive visualization: users see the progression of recursive calls and the propagation of results back to the original call
- Simple syntax: more time learning recursion, less time debugging code
An example using the Fibonacci numbers:

```c
#define recursive int fibonacci (int n):
    base:
       {n == 0} 0
       {n == 1} 1
    recursive:
       { } fibonacci(n-1) + fibonacci(n-2)
```
An example using the Fibonacci numbers:

```plaintext
define recursive int fibonacci (int n):
    base: {
        n == 0} 0
        n == 1} 1
    recursive: {
        fibonacci(n-1) + fibonacci(n-2)
    }
```

Keywords signifying the beginning of the recursive function definition
An example using the Fibonacci numbers:

```c
define recursive int fibonacci (int n):
    base:
        {n == 0} 0
        {n == 1} 1
    recursive:
        { } fibonacci(n-1) + fibonacci(n-2)
```

Return data type
An example using the Fibonacci numbers:

```
define recursive int fibonacci (int n):
    base:
        {n == 0} 0
        {n == 1} 1
    recursive:
        { } fibonacci(n-1) + fibonacci(n-2)
```
Syntax

An example using the Fibonacci numbers:

define recursive int fibonacci (int n):
    base:
    {n == 0} 0
    {n == 1} 1
    recursive:
    { } fibonacci(n-1) + fibonacci(n-2)
An example using the Fibonacci numbers:

```cpp
define recursive int fibonacci (int n):
    base:
        {n == 0} 0
        {n == 1} 1
    recursive:
        fibonacci(n-1) + fibonacci(n-2)
```

Indented block of base cases of the form:
{condition} returnVal
An example using the Fibonacci numbers:

```python
def recursive_fibonacci(n):
    base:
    { n == 0 } 0
    { n == 1 } 1
    recursive:
    { recursive_fibonacci(n-1) + recursive_fibonacci(n-2) }
```
Other syntactical features

- Watch block: comma-delimited list of variables whose values the user wishes to track
- Explicit conditionals (if/else) and loops (for, while)
- Built-in functions for lists
- Print statements and comments
Syntax

Whitespacing

• FIRE uses Python-based whitespace indentation for block delimiting instead of curly braces
• Indentation follows certain statements and signifies the beginning of a block
• Dedentation signifies the end of a block
Advantages

• Makes code more readable
• Easier to write because dedents are automatically matched to indents for the user

Disadvantages

• Harder to implement because indentation levels must be tracked using a stack as opposed to simply detecting matching braces
Stack implementation notes from Python

• For each line, compare indentation level to top of stack (initially contains one pushed 0)
  – If equal, do nothing
  – If larger, push onto stack and issue INDENT
  – If smaller, pop all numbers on stack that are larger and issue DEDENT for each

• On EOF, issue remaining DEDENTS
Screenshot of a recursive call step

BASE: if(3 == 0 || 3 == 1)
FALSE
RECURSIVE: return fib(3 - 1) + fib(3 - 2)

BASE: if(2 == 0 || 2 == 1)
FALSE
RECURSIVE: return fib(2 - 1) + fib(2 - 2)

BASE: if(1 == 0 || 1 == 1)
TRUE

BASE: if(0 == 0 || 0 == 1)
TRUE
Screenshot of a return value propagating back
System Architecture

Overview

User file

ANTLR
InputStream
Characters

Lexer

WSFilter
Tokens

Common
TokenStream

Parser

String
Template
Parsed

Fire.java
Java compiler

Fire.class

JVM

GUI package

Backend

Frontend
System Architecture

Overview

User file

ANTLR InputStream
Characters

Lexer

WSFilter
Tokens

Common TokenStream

String Template

Parser

GUI package

JVM

Fire.class

Fire.java

Java compiler

Backend

Frontend
Overview

System Architecture

- User file
- Error checking + scoping
- ANTLR InputStream
- Characters
- Parser
- String Template
- Parsed Template
- Lexer
- WSFilter
- Common TokenStream
- Tokens
- Fire.java
- Java compiler
- Fire.class
- JVM
- GUI package
- Backend
- Frontend
System Architecture

Overview

User file -> ANTLR InputStream
Characters

Error checking + scoping

Lexer -> WSFilter
Tokens

Common TokenStream

String Template -> Parsed

Parser

Parsed

Fire.java

Java compiler

Fire.class

JVM

GUI package

Backend

Frontend
System Architecture

Overview

User file → ANTLR InputStream → Lexer → WSFilter → Common TokenStream

Error checking + scoping → Lexer → String Template → Fire.java → Java compiler

Parser → String Template → Parsed → Fire.class → JVM

GUI package → Backend → Frontend
System Architecture

Development tools
• Eclipse Ganymede + Subclipse
• ANTLR v3 + StringTemplate 3.2
• Java SE 6 + Swing

Management tools
• Google Code
• Google Docs
Testing

Multi-stage testing approach
• Unit testing whitespacing
• Unit testing grammar production rules
• User testing full examples
## White space testing

<table>
<thead>
<tr>
<th>Source (fibonacci.fire)</th>
<th>Token stream output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>define recursive int fibonacci (int n):</code></td>
<td><code>&lt;DEFINE&gt; &lt;WS&gt; &lt;RECURSIVE&gt; &lt;WS&gt; &lt;TINT&gt; &lt;WS&gt; &lt;NAME&gt; &lt;WS&gt; &lt;LPAREN&gt; &lt;TINT&gt; &lt;WS&gt; &lt;NAME&gt; &lt;RPAREN&gt; &lt;COLON&gt; &lt;NEWLINE&gt;</code></td>
</tr>
<tr>
<td><code>base:</code></td>
<td><code>&lt;INDENT&gt; &lt;BASE&gt; &lt;COLON&gt; &lt;NEWLINE&gt;</code></td>
</tr>
<tr>
<td><code>{n == 0} 0</code></td>
<td><code>&lt;INDENT&gt; &lt;LCURLY&gt; &lt;NAME&gt; &lt;WS&gt; &lt;EQ&gt; &lt;WS&gt; &lt;INT&gt; &lt;RCURLY&gt; &lt;WS&gt; &lt;INT&gt; &lt;NEWLINE&gt;</code></td>
</tr>
<tr>
<td><code>{n == 1} 1</code></td>
<td><code>&lt;LCURLY&gt; &lt;NAME&gt; &lt;WS&gt; &lt;EQ&gt; &lt;WS&gt; &lt;INT&gt; &lt;RCURLY&gt; &lt;WS&gt; &lt;INT&gt; &lt;NEWLINE&gt;</code></td>
</tr>
<tr>
<td><code>recursive:</code></td>
<td><code>&lt;DEDENT&gt; &lt;RECURSIVE&gt; &lt;COLON&gt; &lt;NEWLINE&gt;</code></td>
</tr>
<tr>
<td><code>{ } fibonacci(n-1) + fibonacci(n-2)</code></td>
<td><code>&lt;INDENT&gt; &lt;LCURLY&gt; &lt;WS&gt; &lt;RCURLY&gt; &lt;WS&gt; &lt;NAME&gt; &lt;LPAREN&gt; &lt;NAME&gt; &lt;MINUS&gt; &lt;INT&gt; &lt;RPAREN&gt; &lt;WS&gt; &lt;PLUS&gt; &lt;WS&gt; &lt;NAME&gt; &lt;LPAREN&gt; &lt;NAME&gt; &lt;MINUS&gt; &lt;INT&gt; &lt;RPAREN&gt; &lt;NEWLINE&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;DEDENT&gt; &lt;DEDENT&gt;</code></td>
</tr>
</tbody>
</table>
Testing

Grammar production testing

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRE Program</td>
</tr>
</tbody>
</table>

Tester: Translate program to Java and compare to target code

Output result of comparison

| Equivalent (PASS) | Not equivalent (FAIL) |
Conclusion

Lessons learned

• User-friendly languages are harder to write and test
• Python as a reference language: not everything translates well to Java

Future improvements

• Support for corecusion
• Multiple types of output
Conclusion

Key benefits

• Supports many common recursions of multiple types

• Shallow learning curve; code is both easy to write and understand

• Automated instrumentation of the code and visual output