An Expression-Oriented Approach to Programming Education

Exploiting the Synergy between Computing and Math
Learning Syntax is Known to be an Obstacle in Programming Education

Responses: **Block Coding** (replace syntax with shapes) & **Gradual Languages** (relaxed syntax rules)

Scratch [MIT Media Lab]  Hedy [Felienne Hermans, Leiden University]
Alternative Approach
Exploit Synergy with Math ⇔ Embrace Syntax
Math Abstractions ... Baby Steps

Back to basics - let's reminisce our early computing education:

and I am one baby syntax turtle ➔
Math Abstractions ...

\[ 8192 = 8 \times 1000 + 1 \times 100 + 9 \times 10 + 2 \times 1 \]

\[ \frac{11}{31} + \frac{31}{42} \]

\[ 6 \times 7 = 42 \]

\[ \frac{42}{18} = \frac{21}{9} \]

\[ 9 \times (3 + 4) = 63 \]

\[ 1,000,000 - 997 = 999,003 \]

\[ \frac{2}{7} + \frac{3}{7} = \frac{5}{7} \]
Math Abstractions ...

\[ E = mc^2 \]
\[ y = m \cdot x + q \]
\[ \sin^2 x + \cos^2 x = 1 \]
\[ \frac{\sin x}{\cos x} = \tan x \]
\[ (a - b)(a + b) = a^2 - b^2 \]
\[ D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \]
\[ F = G \frac{m_1 m_2}{d^2} \]
How much syntax should we feed students to start learning programming?

WHAT IF I TOLD YOU ...

THEY ALREADY KNOW THE SYNTAX
They Already Know The Syntax!

<table>
<thead>
<tr>
<th></th>
<th>Raise to Power</th>
<th>Multiplication</th>
<th>Division</th>
<th>Addition</th>
<th>Subtraction</th>
<th>Less than</th>
<th>String Concatenation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATH</td>
<td>( x^y )</td>
<td>( xy \mid x \cdot y )</td>
<td>( x \div y \mid x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x &lt; y )</td>
<td>( xy \mid x \cdot y )</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>( x^{**}y )</td>
<td>( x^{*}y )</td>
<td>( x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x.LT. y )</td>
<td>( x // y )</td>
</tr>
<tr>
<td>LISP</td>
<td>((\text{pow } x \ y))</td>
<td>(* x y )</td>
<td>(/ x y )</td>
<td>(+ x y )</td>
<td>(- x y )</td>
<td>(&lt; x y )</td>
<td>((\text{concatenate } x \ y))</td>
</tr>
<tr>
<td>C / C++</td>
<td>( \text{pow}(x, y) )</td>
<td>( x^y )</td>
<td>( x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x &lt; y )</td>
<td>( x + y )</td>
</tr>
<tr>
<td>Haskell</td>
<td>( x^y \mid x^{**}y )</td>
<td>( x^y )</td>
<td>( x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x &lt; y )</td>
<td>( x ++ y )</td>
</tr>
<tr>
<td>Python</td>
<td>( x^{**}y )</td>
<td>( x^y )</td>
<td>( x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x &lt; y )</td>
<td>( x + y )</td>
</tr>
<tr>
<td>Java</td>
<td>Math.pow(x, y)</td>
<td>( x^y )</td>
<td>( x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x &lt; y )</td>
<td>( x + y )</td>
</tr>
<tr>
<td>JavaScript</td>
<td>( x^{**}y )</td>
<td>( x^y )</td>
<td>( x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x &lt; y )</td>
<td>( x + y )</td>
</tr>
<tr>
<td>OCaml</td>
<td>( x^{**}y )</td>
<td>( x^y \mid x^{*}.y )</td>
<td>( x/y \mid x/.y )</td>
<td>( x + y \mid x+.y )</td>
<td>( x - y \mid x-.y )</td>
<td>( x &lt; y )</td>
<td>( x^y )</td>
</tr>
<tr>
<td>MS-Excel</td>
<td>( x^y )</td>
<td>( x^y )</td>
<td>( x/y )</td>
<td>( x + y )</td>
<td>( x - y )</td>
<td>( x &lt; y )</td>
<td>( x &amp; y )</td>
</tr>
</tbody>
</table>

Note: MATH stands for Generally accepted Math notation

::: OK, sure, but those are just expressions. Computing them is not programming, right? Expressions aren’t enough, right? ➔
“It may seem odd to describe a spreadsheet as a programming language. Indeed, one of the great merits of spreadsheets is that users need not think of themselves as doing “programming”, let alone functional programming — rather, they simply “write formulae” or “build a model”. However, one can imagine printing the cells of a spreadsheet in textual form, like this:

\[
\begin{align*}
A1 &= 3 \\
A2 &= A1 - 32 \\
A3 &= A2 \times \frac{5}{9}
\end{align*}
\]

and then it plainly is a (functional) program.”

to program with spreadsheets all the syntax you need is that of expressions!
Critique of the Traditional Spreadsheet Core

- **Lack of functional abstraction**
  - Considerable research work has been done on this
  - December 3rd, 2020: Microsoft Research announced LAMBDA

- **Overly simplistic type system**
  - All top-level variables must be a worksheet
  - Worksheets are non-composable cell containers
  - All cells are unitype and must be referenced via coordinates
  - A1 notation should be considered harmful

- **Entanglement of model and visualization**
  - Worksheets are the only true variables of the core
    - They are containers that hold state, which includes unreduced expressions
  - Worksheets are also the primary element of the presentation
    - They play an important role as UI layout managers

A language-centric redesign of spreadsheets has been shown to work

**ZenSheet / Lilly**

ZenSheet supports functional abstraction and composable data structures. 2D arrays can be used as worksheets: it truly generalizes spreadsheets!
ZenSheet: a live programming environment for reactive computing

Who: Enzo Allo, Monica Figuera
Track: LIVE 2017
When: Tue 24 Oct 2017 15:30 - 15:50 at Regency D - Winter
Abstract: We introduce ZenSheet: an experimental live programming environment for reactive computing. We emphasize the most relevant design and implementation choices, and provide a glimpse about future work. The ZenSheet project aims to generalize spreadsheets in an intuitive way. It implements a superset of the core functionality of traditional spreadsheets, adding concepts from modern programming languages. The end purpose is to make spreadsheet computing valuable to an even wider audience spectrum.

File attachments:
- PDF Preprint (ZenSheet.pdf) 475KB

Authors:
- Enzo Allo, Lakeboli Research, United States
- Monica Figuera, Universidad Simon Bolivar
### Functional Programming for End-Users

**Enzo Alda, Javier López**

```plaintext
W1B3:C11:= formula(W1B2:C2);
```
ENZO ALDA

functional programming for everyone!

https://www.youtube.com/watch?v=mJa0_gKE6xo
**Higher-Order Functions**

---

```
sys.cycles(0);
```
Rich Type System: Dynamic Arrays, Tuples, Structs
Wide-Spectrum Computing
Experience Report: Expressions-First Programming 101
Experience Report: Modernizing Programming 101 with an Expressions First Approach

• Setting

  • Location: Simón Bolívar University – Venezuela
    • Ranked top 5 among Latin American universities in the 80s
  • Course: Programming 101 for engineers, excluding software engineering students
    • Nearly all students have no programming experience
  • Delivery: interactive online (programming theory & lab) plus recorded video
    • Only 10 usable weeks (20 lectures) for theory and 8 usable weeks (16 hours) for lab instruction
    • Students set up their own lab, with material and some technical assistance from the lab instructor
    • Key tools: Git & GitHub (https://github.com/), MSYS2 (https://www.msys2.org/), code editors
Experience Report:
Modernizing Programming 101 with an Expressions First Approach

• Objectives

• **Old** course objectives:
  • students must learn basic C programming
  • emphasis is placed on array processing and I/O

• **New** course objectives: form students who exhibit
  1. A clear understanding of **basic programming concepts**
  2. Confidence in their **ability to learn other languages**
  3. **Basic proficiency** in one or more languages
## Expressions

<table>
<thead>
<tr>
<th>Math</th>
<th>C++</th>
<th>JavaScript</th>
<th>Lilly</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a(c + b) = ac + ab)</td>
<td>(a \ast (b + c) == a\ast c + a\ast b)</td>
<td>(a \ast (b + c) == a\ast c + a\ast b)</td>
<td>(a \ast (b + c) = a\ast c + a\ast b)</td>
</tr>
<tr>
<td>(\frac{1}{(1 + \frac{1}{n})^n})</td>
<td>(1 / \text{pow}(1 + 1.0/\text{n}, \text{n}))</td>
<td>(1 / (1 + 1/\text{n})^{\text{n}})</td>
<td>(1 / (1 + 1/\text{n})^\text{n})</td>
</tr>
<tr>
<td>(\pi r^2)</td>
<td>(\text{M_PI} \ast \text{pow}(r, 2))</td>
<td>(\text{Math_PI} \ast r^{\ast 2})</td>
<td>(\text{pi}() \ast r^{\ast 2})</td>
</tr>
<tr>
<td>(\sin^2 x)</td>
<td>(\text{pow}(\sin(x), 2))</td>
<td>(\text{Math_sin}(x)^{\ast 2})</td>
<td>(\sin(x)^{\ast 2})</td>
</tr>
<tr>
<td>(\sin x^2)</td>
<td>(\sin(\text{pow}(x, 2)))</td>
<td>(\text{Math_sin}(x^{\ast 2}))</td>
<td>(\sin(x^{\ast 2}))</td>
</tr>
<tr>
<td>(\sin \sin x)</td>
<td>(\sin(\sin(x)))</td>
<td>(\text{Math_sin}(\text{Math_sin}(x)))</td>
<td>(\sin(\sin(x)))</td>
</tr>
</tbody>
</table>
### Functional Abstraction
dynamically typed languages

<table>
<thead>
<tr>
<th>Math</th>
<th>JavaScript</th>
<th>Lilly</th>
</tr>
</thead>
</table>

\[
A = \pi r^2
\]

\[
D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}
\]

let area = \(r \rightarrow \text{Math.PI} \times r^2\);

let point1 = \{x: 3, y: 2\};
let point2 = \{x: 6, y: 6\};

function distance (p1, p2) {
  return Math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2)
}

distance(point1, point2)

:: area := fn(r) -> pi() * r^2;

type Coord = struct { x; y; };

:: point1 = Coord(3, 2);
:: point2 = Coord(6, 6);

:: distance := fn (p1, p2) ->
  sqrt((p1.x - p2.x)^2 + (p1.y - p2.y)^2);

distance(point1, point2)
# Functional Abstraction

**statically typed language**

<table>
<thead>
<tr>
<th>Math</th>
<th>C++</th>
<th>Lilly</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ A = \pi r^2 ]</td>
<td>[ D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} ]</td>
<td>[ :: area := fn(double r) =&gt; double -&gt; pi() * r^2; ]</td>
</tr>
</tbody>
</table>
| double area(double r) {  
|     return M_PI * pow(r, 2);  
| } | type Coord = struct {  
|     double x;  
|     double y;  
| };  
| Coord point1 = {3, 2};  
| Coord point2 = {6, 6};  
| double distance (Coord p1, Coord p2) {  
|     return sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));  
| } | :: point1 = Coord(3, 2);  
| :: point2 = Coord(6, 6);  
| :: distance := fn (Coord p1, Coord p2) => double -> sqrt((p1.x - p2.x)^2 + (p1.y - p2.y)^2); |
Inductive Math definitions => Recursion

```cpp
// fibonacci
/// fibo(0) ==> 0
/// fibo(1) ==> 1
/// fibo(n) ==> fibo(n-1) + fibo(n-2) // ... for n > 1

// C++
int fibo(int n) {
    return n < 2 ? n : fibo(n-1) + fibo(n-2);
}
```
```javascript
// JavaScript
let fibo = n => (n < 2 ? n : fibo(n-1) + fibo(n-2));
```
```lilly
// Lilly (dynamically typed)
:: fibo := fn(n) -> if(n < 2, n, fibo(n - 1) + fibo(n - 2));
```
```lilly
// Lilly (statically typed)
:: fibo := fn(int n) => int -> if(n < 2, n, fibo(n - 1) + fibo(n - 2));
```
Expressions-First Course Plan

Level 0
- Values and basic types
- Computing expressions using literal constants
- Variable definitions: using variables in expressions
- Computing with simple values and tuples
- Defining functions by abstracting expressions

Level 1
- Making algorithmic decisions: conditional expressions
- Defining recursive functions from inductive definitions
- Computing functional reductions over sequences
- Understanding tail recursion and iteration

Level 2
- Higher order functions
- Curry transformations
- Nested functions and scope rules

Level 3
- Pointers and references
- Associative maps and mutable arrays
- Persistence

Level 4
- Recursive types: trees
- Sum types: dealing with polymorphism
- Polymorphism in dynamically typed languages
- Polymorphism in statically typed languages
- Polymorphism in functional languages
Experience Report:  
Modernizing Programming 101 with an Expressions First Approach

Findings

So far, students have shown:

1. a practical understanding of the **difference between dynamically typed and statically typed languages**, well beyond a merely abstract notion, having been exposed to JavaScript, C++, and Lilly for several weeks;

2. a good grasp of the **difference between the functional and imperative programming styles**, due to spending the first 4 weeks without performing any state mutation besides defining and initializing immutable variables, essentially using an SSA-form (single static assignment) style of programming in JavaScript, C++, and Lilly;

3. ability to **code in JavaScript and C++**, completing programming assignments for the levels covered so far.
References


Thank You!