Hippograph

The Language for High Performance Parsing of Graphs
Motivation
Graph theory is an important field in computer science, with wide ranging applications.

We thought there should be a language that made experimenting with and utilizing graphs easier!

giraph from Fall 2017 was a major inspiration for us, but we had some ideas for what could be added...
Goals

1. **Unified graph type** - generic graph type that can handle any type of edge
2. **Customizable node names** - giving the user greater control over their graphs
3. **Cypher-like query capabilities** - especially helpful when using graph to store large amounts of data
4. **Anonymous functions** - for passing in user-defined graph operations
5. **Search Strategy Type** - specifying traversal method in graph iteration
Workflow and Team Processes
The end result

<table>
<thead>
<tr>
<th>File</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanner.mll</td>
<td>70</td>
</tr>
<tr>
<td>parser.mly</td>
<td>160</td>
</tr>
<tr>
<td>ast.ml</td>
<td>178</td>
</tr>
<tr>
<td>sast.ml</td>
<td>109</td>
</tr>
<tr>
<td>semant.ml</td>
<td>446</td>
</tr>
<tr>
<td>codegen.ml</td>
<td>823</td>
</tr>
<tr>
<td>graph.c</td>
<td>1,152</td>
</tr>
<tr>
<td>hippograph.ml</td>
<td>29</td>
</tr>
</tbody>
</table>

Plus

197 Test Scripts

156 Git Commits

2 Pies of Pizza
Language Overview
The Basics

● Operators:
  ○ + - * / ; = . > < => <= == and or not

● Control Flow:
  ○ While (true) {make_graphs();}
  ○ For (int i = 0; i <= 10; i = i + 1)
  ○ If (you_dont_mind()) { do_it(); } else { dont_bother(); }
    ■ The ELSE clause is optional!

● Primitive Types:
  ○ int, bool, string

● Comments:
  ○ (* don’t run me! *)
Function Flavors

The Standard:

```c
return_type func_name(type1 arg1; type2 arg2; ... ) {
    ...
}
```

The Condensed:

```c
fun<type1:type2: ... :typek, ret_typ> f = ret_type (type1 ... )( expr )
```
The Condensed Function

- Allow declarations of functions within the bodies of other functions
  - Stored in variables, which effectively provide the names of anonymous functions
  - Fall in and out of scope with the function!

- Implemented as expressions which resolve to a \texttt{FUN} type

- Passing functions as first class data: WIP.
What about graphs?

- **Node Expressions:**

  \[
  \text{Node}\{t_1:t_2\} = \text{expr}\_\_\_\_\_t_1 : \text{expr}\_\_\_\_\_t_2; \\
  \text{Node}\{t_1:t_2\} = \text{expr}\_\_\_\_\_t_1; \\
  \text{Node}\{t_1\} = \text{expr}\_\_\_\_\_t_1;
  \]

- **Graph Expressions:**

  \[
  \text{Graph}\{\text{int}:\text{bool}, \text{int}\} = [1:\text{true} <(5)> 3 <(3)- 8:\text{true}; 8 -(4)- 1]; \\
  \text{Graph}\{\text{int}\} = [1 <()> 3 <()- 8; 8 -() - 1];
  \]
Implementation
Architecture

Scanner → Parser → Semantic Checker → Code Generation

Hippograph

Executable

LLVM

Graph.c
Graphs

Implemented as adjacency lists

Union primitive allowed for flexible typing.

Under the hood, all edges are directed. Non-directional and bidirectional edges are implemented as two one-way edges.
Semantic Checking

```
and check_graph_expr #dec1s vars node_list edge_list -
(* infer node label/data types from first nodes in list if any,
and check that all items have the same type *)
let node_label_typ, node_data_typ, s_node_list =
  if node_list = []
    then (Bool, Bool, []) (* bool type, for now *)
  else let err = "type mismatch in graph nodes in"
    let check_node_typ (lt_opt, dt_opt) n =
      match n with
      | (Node(lt, dt), SNodeExpr(_, d)) ->
        (* check matching node label *)
        let lt_opt = (match lt_opt with
          | None -> Some(lt)
          | Some(lt') -> if lt = lt'
            then lt_opt
            else raise (Failure err)) in
        (* check matching node data *)
        let dt_opt = (match d with
          | (Bool, SNull) -> dt_opt
          | _ -> match dt_opt with
            | None -> Some(dt)
            | Some(dt') -> if dt = dt'
              then dt_opt
              else raise (Failure err))
        in (lt_opt, dt_opt)
      | _ -> raiseUnsupported_constructor
```
For every new feature implemented, a small test was created to ensure it worked as expected.
Demo

Bellman-Ford Algorithm
graph<string:int, int> g = ["S":500 -(10)> "A":500 -(2)> "C":500 -(2)> "B":500 -(1)> "A"; "S" -(8)="E":500 -(1)> "D":500 -(1)>"C"; "D" -(4) > "A"];

Initial Graph
Shortest-path Graph

Original Graph:
"S" : 500 -> ["A" : 500 (10), "E" : 500 (8)]
"A" : 500 -> ["C" : 500 (2)]
"C" : 500 -> ["B" : 500 (2)]
"B" : 500 -> ["A" : 500 (1)]
"E" : 500 -> ["D" : 500 (1)]
"D" : 500 -> ["C" : 500 (1), "A" : 500 (4)]

Shortest Path:
"S" : 0 -> ["A" : 10 (10), "E" : 8 (8)]
"A" : 10 -> []
"C" : 10 -> ["B" : 12 (2)]
"B" : 12 -> []
"E" : 8 -> ["D" : 9 (1)]
"D" : 9 -> ["C" : 10 (1)]
Negative Edge Weight Cycles in Graph

Original Graph:
"S":500 -> ["A":500 (10), "E":500 (8)]
"A":500 -> ["C":500 (-7)]
"C":500 -> ["B":500 (2)]
"B":500 -> ["A":500 (1)]
"E":500 -> ["D":500 (1)]
"D":500 -> ["C":500 (1), "A":500 (4)]

Negative edge weight cycle
Thank you!

Special thanks to our TA Jennifer “codejen.ml” Bi!