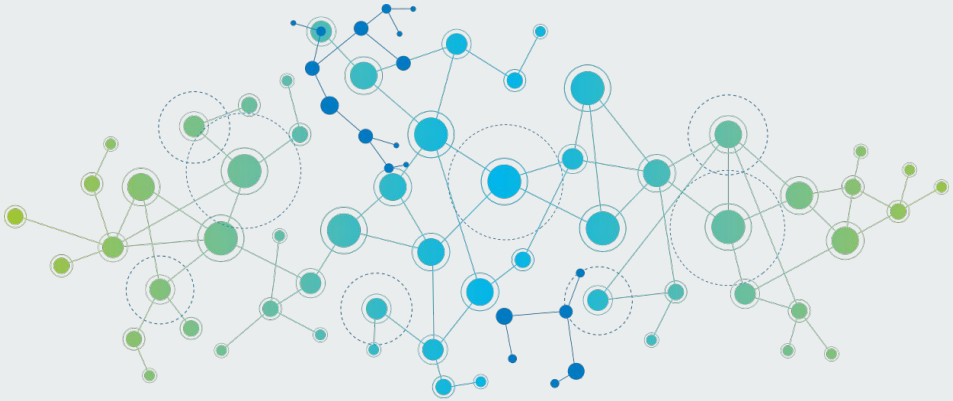
YAGL

Yet Another Graphing Language

Adam Carpentieri (ac4409)
James Mastran (jam2454)
Jack Hurley (jth2165)
Shvetank Prakash (sp3816)



Final Presentation

Meet The Team

Adam Carpentieri

Manager

James Mastran

Language Guru

Shvetank Prakash

System Architect

Jack Hurley

Tester

Agenda

1. Target Audience & Motivation
2. YAGL In One Slide
3. Compiler Architecture
4. Cool YAGL Components
5. YAGL Standard Library
6. Built-in Functions (C)
7. Who Did What?
8. Demo
9. Q&A

About YAGL: Target Audience & Motivation

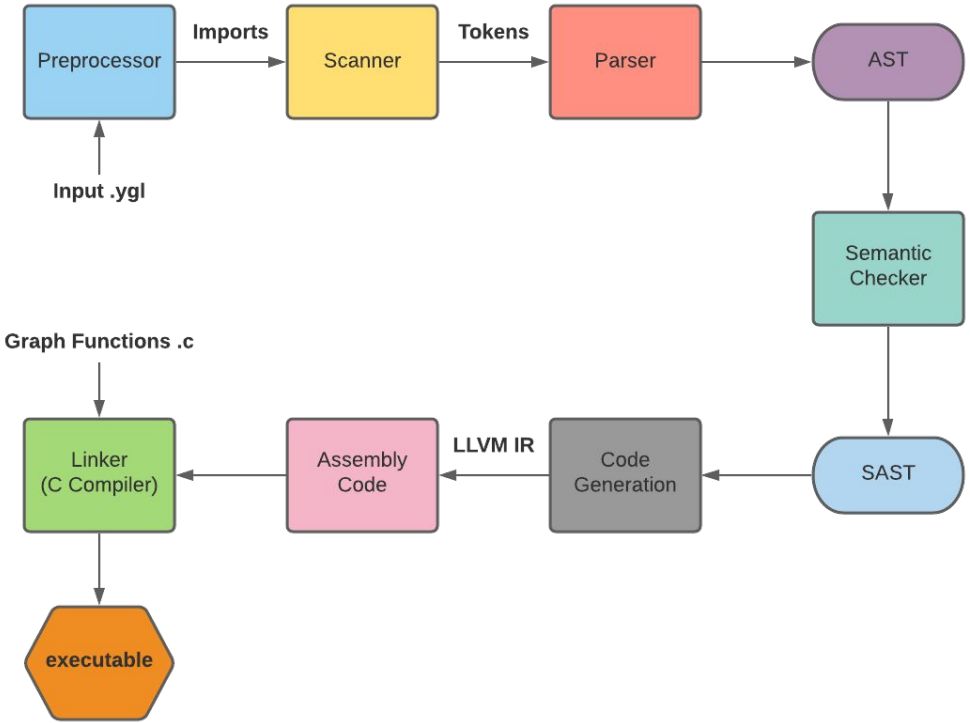
- Pervasiveness of graphs in CS ⇒ great candidate
  - fundamental in data structures & algorithms
- Aims to make implementing graphs & algorithms much simpler
- Ubiquitous with numerous applications:
  - social media connections
  - roads that connect cities
  - flights between cities
  - many other mathematical & logical problems
- Statically & Strongly typed, imperative language
- C-like syntax but adopted other languages features we appreciated

YAGL in One Slide

- No `main()`
- Import standard library for `print_graph_lib`
- Declaring and initializing graphs and nodes (more later)
- Adding nodes and edges to graphs
- Scoping

```
1  import stdgraph.ygl
2
3  Graph example;
4  Node n(gen_name("1"));
5  Node n2(gen_name("2"));
6
7  example: + n + n2, n -> n2 -> n;
8
9  print_graph_lib(example);
10
11 String gen_name(String n) {
12         String hidden = "ODE";
13         String total = "";
14         {
15                 String hidden = "N";
16                 total = total + hidden;
17         }
18         return total + hidden + n;
19 }
```

Compiler Architecture



Cool YAGL Features

No `main()`

- Each file has an implicit main function
  - Entry point

- Implemented via lifting all “orphaned” statements
  - Statements not within a function

```
printString("Hello World!");
```

Generic printing capabilities

```
1   Node n("hi");
2   print(n);
3   int x = 5;
4   print(x);
5   char s = 's';
6   print(s);
7   print('s');
8   Graph g;
9   g: + n;
10  print(g);
11  print("Hi!!!");
12  print(true);
```

```
in List.fold_left add_bind StringMap.empty [ ("print", Void);
```

```
| SCall ("print", [x]) -> (
            match x with
                (Node, _) -> (L.build_call print_node_func [| expr builder s_table x |] "print_node" builder)
              | (Graph, _) -> L.build_call print_graph_func [| expr builder s_table x |] "print_graph" builder
              | (Int, _) | (Bool, _) -> L.build_call printf_func [| int_format_str ; (expr builder s_table x) |] "printf" builder
              | (Char, _) -> L.build_call printf_func [| char_format_str ; (expr builder s_table x) |] "printf" builder
              | (Float, _) -> L.build_call printf_func [| float_format_str ; (expr builder s_table x) |] "printf" builder
              | (String, _) -> L.build_call printf_func [| string_format_str ; (expr builder s_table x) |] "printf" builder
              | _ -> raise (Failure("Not implemented print type.")))
```

Arrays

- Different syntax than C
  - int[10] foo vs int foo[10]
- Flexible in ways it can be used and accessed
  - Arrays of all types and any [expr] inside
- LLVM `getelementptr` understanding key

```
let indices =
  (Array.of_list [L.const_int i64_t 0; index]) in
let ptr =
  L.build_in_bounds_gep (lookup s s_table) indices (s^"_ptr_") builder
in L.build_store e' ptr builder
```

```
int bar;
bar = 9;

int[10] foo;
foo[0] = 0;
foo[2+3] = 123;
foo[bar] = 456;

printInt(foo[3-3]);
printInt(foo[5]);
printInt(foo[bar]);

int temp;
temp = foo[0];
printInt(temp);
```

Scoping

- Each block has his own scope
  - C-like scoping rules
- Variables not just declared at top
- Implemented via symbol tables
  - Semantic Checker & Codegen pass around a list of symbol tables

```
/* Every single variable is named foo */

String foo = "wins";
{
   String foo = "always";
   {
      bool foo = true;
      {
         if(foo) {
            int foo = 42;
            printInt(foo);
         }
      }
   }
   printString(foo);

}
printString(foo);
```

Preprocessing / Importing

- `import` keyword
  - Python inspiration
  - Acts similar to C's preprocessing directives
- File imported is "pasted" to provide access to all functions and vars
- Done prior to feeding to scanner

```
import stdalgo.ygl
import stdgraph.ygl

/* Social Media Application */

Graph fb;         /* Facebook  */
Graph tw;         /* Twitter   */
```

•

•

•

```
/* Find friends of friends */
dfs(fb, james, 2);           /* S
print(fb);
```

Graphs with Nodes & Edges

- Allocates room for empty Graph on the heap
  - Graphs dynamically grow to hold “infinite” nodes
- Allocates and initializes a node with given name
- Adds Node to the team Graph
  - Nodes can be placed in multiple graphs
- Recursively adds Edges to graph with default weight of 1

```
1
2   Graph team;
3
4   Node adam("Adam");
5   Node james("James");
6   Node tank("Tank");
7   Node jack("Jack");
8
9   team + adam;
10  team + james;
11  team + tank;
12  team + jack;
13
14  team: adam -> james -> tank -> jack -> adam;
15
16  printGraph(team);
```

```
Nodes:
Node (0) : Adam --- Node (1) : James --- Node (2) : Tank
Node (3) : Jack

Edges:
[(0) : Adam] ---(1)---> [(1) : James]
[(1) : James] ---(1)---> [(2) : Tank]
[(2) : Tank] ---(1)---> [(3) : Jack]
[(3) : Jack] ---(1)---> [(0) : Adam]
```

More Complex Augmentations

- Complicated example to show variety of graph operation in single LOC
  - Add Node +
  - Add Edge -> and <-
  - Add Bidirectional Edge <->

```
Graph demo;

Node portland("Portland");
Node pittsburgh("Pittsburgh");
Node nyc("New York City");
Node nj("New Jersey");

demo: + portland + pittsburgh ->2566 portland ->2566 pittsburgh
      + nyc [800<->800] pittsburgh, portland [2900<->2900] nyc,
      + nj <-13 nyc <-13 nj [2899<->2899] portland,
      nj [790<->790] pittsburgh;

print(demo);
```

```
Nodes:
Node (0) : Portland --- Node (1) : Pittsburgh --- Node (2) : New York City
Node (3) : New Jersey

Edges:
[(1) : Pittsburgh] ---(2566)---> [(0) : Portland]
[(1) : Pittsburgh] ---(800)---> [(2) : New York City]
[(1) : Pittsburgh] ---(790)---> [(3) : New Jersey]
[(0) : Portland] ---(2566)---> [(1) : Pittsburgh]
[(0) : Portland] ---(2900)---> [(2) : New York City]
[(0) : Portland] ---(2899)---> [(3) : New Jersey]
[(2) : New York City] ---(800)---> [(1) : Pittsburgh]
[(2) : New York City] ---(2900)---> [(0) : Portland]
[(2) : New York City] ---(13)---> [(3) : New Jersey]
[(3) : New Jersey] ---(13)---> [(2) : New York City]
[(3) : New Jersey] ---(2899)---> [(0) : Portland]
[(3) : New Jersey] ---(790)---> [(1) : Pittsburgh]
```

Accessors

- Graphs
  - graph.weight[node1, node2]
  - graph.num_nodes
  - graph.num_neighbors[A]
  - graph.node[n]
  - graph.neighbor[A, n]
- Nodes
  - v.name
  - v.curr_dist
  - v.visited
- Strings
  - string.length

```
| "num_nodes"          ->      let e' = expr builder s_table (Node, sId) in
                               ignore (L.set_alignment 4 e');
                               let ptr = L.build_struct_gep e' 1 "get_num_nodes" builder in
                               L.build_load ptr "num_nodes_ptr"
```

YAGL's Standard Libraries

- stdgraph.ygl
  - Graph copy_graph_lib(Graph g)
  - Graph reverse_graph_lib(Graph g)
  - void print_graph_lib(Graph g)
- stdalgo.ygl
  - void dfs(Graph G, Node vertex, int depth)
  - Node get_first_node_at_depth(Graph G, Node vertex, Node break, int depth)

Used in upcoming demo!

```
Node get_first_node_at_depth(Graph g, Node vertex, Node b, int depth) {
        reset(g);
        return get_first_node_at_depth_helper(g, vertex, b, depth);
}

Node get_first_node_at_depth_helper(Graph g, Node vertex, Node break, int depth){
   if (vertex.visited == true) {
       return break;
   }

   if (depth == vertex.curr_dist) {
       return vertex;
   }
   vertex.visited = true;

   int size = g.num_neighbors[vertex];
   int curr = 0;

   while (curr < size) {
      Node current;
      current = g.neighbor[vertex, curr];
      if (current.curr_dist == 0) {
          current.curr_dist = vertex.curr_dist + 1;
      }
      curr = curr + 1;
   }
   curr = 0;
   Node new; /* just a place holder */
   while (curr < size) {
      Node current;
      current = g.neighbor[vertex, curr];
      new = get_first_node_at_depth_helper(g, current, break, depth);
      if (new == break) {
      } else {
        return new;
      }
      curr = curr + 1;
   }
   return new;
}
```

Built-in Functions

- Graph Functionality
  - make_graph(int size)
  - insert_node(struct Graph *, struct Node *)
  - make_node(char *name)
  - get_neighbor(struct Graph *, struct Node *)
  - print_graph(struct Graph *)
  - insert_edge(struct Graph *, struct Node *, int, struct Node *)

Who did what?

- First half through Hello World: Together
- Second half: Distributed Feature Development
  - From scanner → … → codegen → tests
- Weekly meeting to merge code & features
- Process worked very well: all wanted to learn about the entire compiler!

Amazing Demo Time

Buckle up.

Q&A

Thank you!