nodable

Karen Shi	Manager
Ajita Bala	Language Guru
Ariel Goldman	System Architect
Naviya Makhija	Tester

Agenda

1. nodable overview
2. Compiler Architecture
3. Lists
4. Nodes
5. Graphs and Trees
6. Future work
7. Demonstrations

nodable overview

- nodable is an imperative, statically typed graphing language designed to help users create, use, and manipulate graphs and trees
- nodable syntax is based on C and Java

- Graphs are used to represent relationships in data and have multiple real-life uses, including modeling social networks, contact tracing, and finding the shortest path in a network
- Our language aims to simplify graphs by providing built-in functions and data structures useful for commonly-used tree and graph algorithms

primitive types

- nodable has four primitive types: **int, float, boolean**, and **string**.
- our language can support **int**, **float**, **boolean**, and **string** literals
  - **integer** literals are sequences of multiple decimal digits
  - **float** literals are sequences of decimal digits
  - **boolean** literals are either “true” or “false”
  - **string** literals are character sequences enclosed in double quotes

identifiers and variables

- Identifiers are names that the user can give to functions and variables.
- Identifiers can consist of a combination of uppercase letters, lowercase letters, digits, and underscores, but must begin with an uppercase or lowercase letter

```
id = ['a' - 'z' 'A' - 'Z'] ['a' - 'z' 'A' - 'Z' '0'-'9' '_']*
```

- nodable variables are instantiated by stating the data type of the variable followed by its identifier. They can also be initialized with a value upon instantiation:

```
int a;

int b = 10;
```

- Variables can be global or local (declared within a function)

```
boolean i;
int main()
{
  int i;
  i = 42;
  print(i + i);
  return 0;
}
```

functions

- All nodable programs must contain a main() function in order to execute correctly, as main() is the entry point of the program
- The user can declare and write their own functions that can be called in main.
  - Function declaration syntax:

    ```
    return_type function_name (params) {...}
    ```

  - Functions can return any of the primitive data types, void, nodes, or lists

operators

- Nodable has 5 categories of operators - arithmetic, unary, assignment, relational, and logical operators

Type of operator	Examples
Arithmetic	+, -, *, /, % (all left-associative)
Unary	!
Assignment	= (right-associative)
Relational	>, <, ≤, ≥, == (all left-associative)
Logical	&&, || (non-associative)

Control Flow

IF/ELSE

```
int x;
x = 10;
if (b)
 if (x == 10)
  x = 42;
else
 x = 17;
return x;
```

WHILE

```
int j;
j = 0;
while (a > 0) {
 j = j + 2;
 a = a - 1;
}
return j;
```

FOR

```
 for (i = 0 ; i < 5 ;
i = i + 1) {
  print(i);
 }
```

lists

- nodable has two fundamental data structures: **lists** and **nodes**
- lists are mutable collections of objects or primitive data types or of lists and nodes. Users can instantiate empty lists or filled lists, and can later append, update:

```
list<int> a = [1, 2, 3];
append(a, 4); //appends 4 to the end of the list
a = update_elem(7, a, 0); //replaces element at index 0 with 7
//a = [7, 2, 3, 4];
print(size(a)); //4
```

- lists can be nested as well:

```
list<list<int>> b = [[1, 2, 3], [2, 3, 1], [9, 8], []];

list<node<int> > t; = [$1, $2, $3];
```

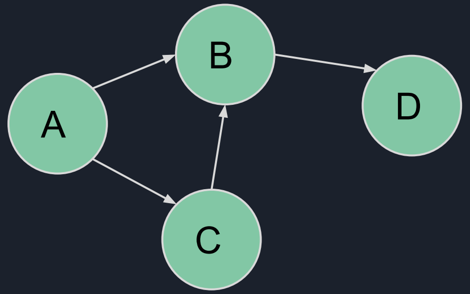
nodes

- Nodes are the other fundamental data type in nodable. Nodes have a unique identifier, a data field, and can have left or right children
- Nodes can have any of the four primitive data types as children - ints, floats, booleans, or strings. They must be declared as one of these four types upon instantiation
- Data literals are assigned to nodes using the **$** symbol
- The built-in functions add_left and add_right are used to create a parent-child edge, and get_left and get_right can be used to access the child nodes

```
node<string> n1;
node<int> n2;
node<float> n3;

n1 = $”i am a string node!”;
n2 = $4;
n3 = $3.14;

add_left(n1, n2);
add_right(n2, n3);

print(get_left(n1).data); //4
printf(get_right(n2).data); //3.14
```

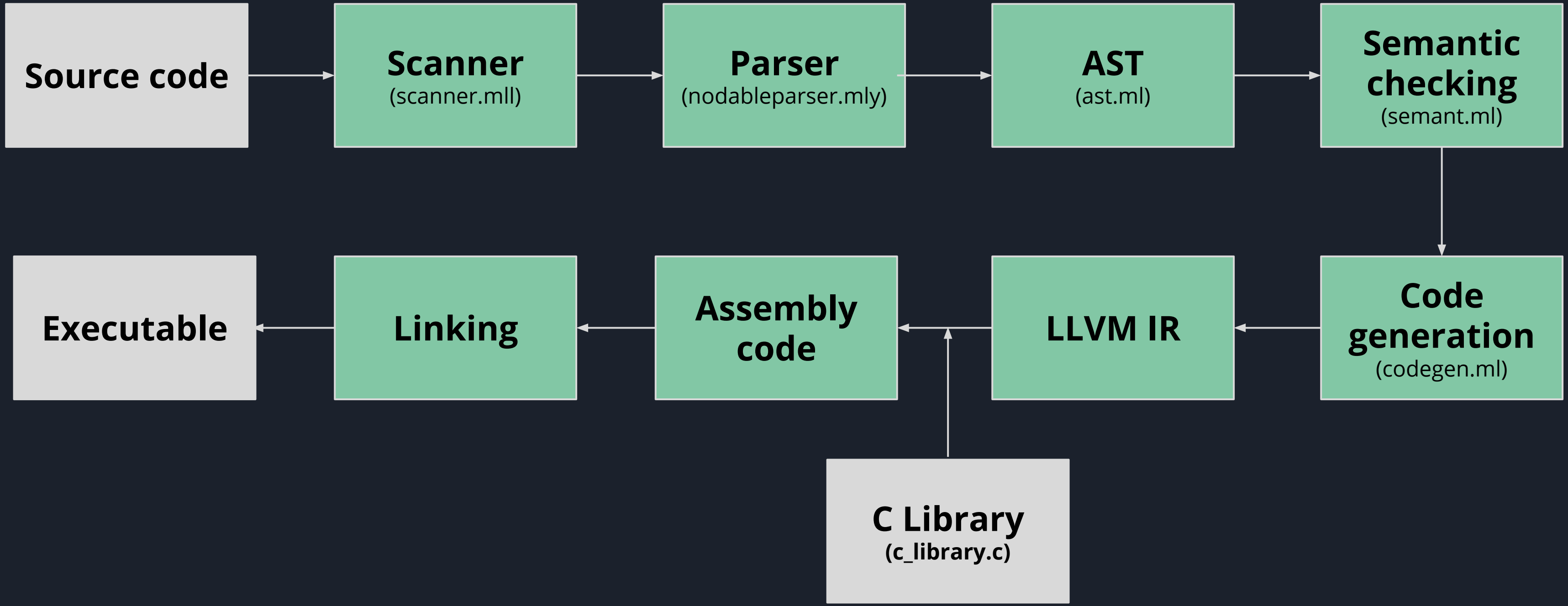
graphs and trees

- nodable does not have data types for graphs and trees. However, these data structures can be represented through nodes and lists
- binary trees can be easily implemented through the usage of the node’s get_**left** and get_**right** attributes
- graphs can be represented using a list of nodes and an adjacency list, as seen in the example on the right
- weighted graphs can also be represented using a list of lists of lists of ints



```
node<string> a = $“A”;
node<string> b = $“B”;
node<string> c = $“C”;
node<string> d = $“D”;

list<node<string>> nodelist
= [a, b, c, d];
list<list<int>> adjlist =
[[1, 2], [3], [1], []];
```

Compiler architecture



Source code

Scanner
(scanner.mll)

Parser
(nodableparser.mly)

AST
(ast.ml)

Semantic checking
(semant.ml)

Code generation
(codegen.ml)

LLVM IR

Assembly code

Linking

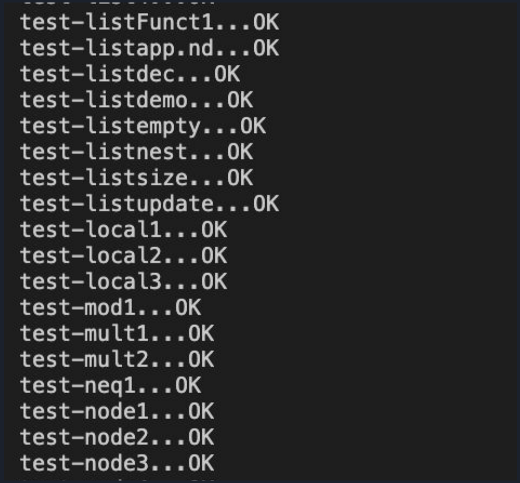
Executable

C Library
(c_library.c)

Testing

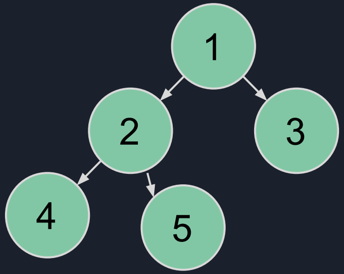
- Automated testing using testall.sh
  - Script that iterates through /tests
  - *.diff with .nd and .out files
  - 120+ test files
- Fail Tests
  - Checked for failure tests that gave the error messages we were expecting to help users debug

- Tested each operator, variable, functions, etc.
- Created more comprehensive tests that implemented many features together



```
test-listFunct1...OK
test-listapp.nd...OK
test-listdec...OK
test-listdemo...OK
test-listempty...OK
test-listnest...OK
test-listsize...OK
test-listupdate...OK
test-local1...OK
test-local2...OK
test-local3...OK
test-mod1...OK
test-mult1...OK
test-mult2...OK
test-neq1...OK
test-node1...OK
test-node2...OK
test-node3...OK
```

emonstrations



1. List manipulation
   a. Declare a list of node<int> elements, and get its size and the average of its values
   b. Reverse the list
   c. Sort the list using selection sort
2. Tree Traversal
   a. Declare nodes, as well as their left and right children nodes
   b. Recursively perform a preorder, postorder, and inorder traversal on the trees and print the node values
3. Check Tree Balance
   a. Declare nodes, as well as their left and right children nodes
   b. Uses recursive tree height function to determine the height of left and right subtrees
   c. Recursively compare heights of subtrees until leaf nodes are reached

Future work

- Implement trees and graphs as actual data structures
    - Include in each graph a list of nodes and an adjacency list for edges
- Allow users to check if a node is null rather than reserving the value 0 for null nodes
- Prevent users from breaking tree rules by adding error warnings