Graphene
Intro/Motivation

- In most programming languages, when a user needs to utilize a graph-like data type to perform an operation, they must expend non-trivial effort writing their own types to represent such graphs and functions to utilize these.
- Graphene intends to be a small, C-like language designed to alleviate this annoyance. Graphene has C-like syntax with flexible built-in operators, types, and functions that allow users to easily create graphs and implement a wide variety of graph algorithms.
- We started with microC’s compiler as our foundation and added/changed as needed.
Features - Primitive Types

- **int** - Standard 32 bit integer type, integers act as booleans as they do in C, 0 = false, nonzero = true
  - int x = 23;
  - 23;

- **float** - Standard double floating point type.
  - float f = 1.1;
  - 1.1;

- **string** - Immutable sequence of 8-bit characters, enclosed in double quotes.
  - string s = “string”;
  - “string”

- ints and floats are compared and passed by value, strings are compared by value and passed by reference.
Features - Built-In Types - List

Lists, declared with “list<t>”, are linked lists than can store any other Graphene type

```cpp
list<int> l;
```

- Elements can be pushed to the front or back of lists
  ```cpp
  l.push_back(20);
  ```
- Lists can be indexed using the [] operator
  ```cpp
  l[0]; // = 20
  ```
- Lists include pop_front/back functions, peek_front/back functions, and a size field
  ```cpp
  l.size; // = 1
  l.peek_front(); // = 20
  l.pop_front(); // = 20
  l.size; // = 0
  ```
Features - Built-In Types - Node

Nodes are wrapper types that can wrap any primitive Graphene type.

```cpp
node<string> n;
```

- **Type wrapped by node cannot be changed, but the value can be reassigned.**
  - `n.val = "node"; // n.val stores a reference to "node"

- **Nodes contain an integer id (used in graph type) and contain a list of edges.**
  - `n.id = 2;`
  - `n.edges.size; // = 0`

- **Nodes are passed and compared by reference, a node variable can be reassigned to reference a different node wrapping the same type.**
  - `node<string> m;`
  - `n == m; // 0`
  - `n = m;`
  - `n == m; // 1`
**Features - Built-In Types - Edges**

Edges are wrapper types that contain a weight (of wrapped type), a destination node (wrapping the same type as the edge), and can be non-traversable.

- Edges are declared using special operators on nodes, with a default “weight” of 0 (or 0.0 or “”) unless a weight is specified with [ ].
- Edge fields cannot be reassigned after initialization, but they can be accessed.
  - `e.weight; // wrapped type of e`
  - `e.dest; // reference to node`
  - `e.t; // 1 if traversable, 0 if not`
Features - Built-In Types - Edge Operators

- Edge operators initialize all three fields of edges
  - node<int> n;
  - node<int> m;
  - n -> m; // directed edge<int> from n to m, weight = 0 (default)
- The above operation creates two copies of the same edge, one traversable, the other not, and stores them accordingly in both nodes’ edge lists.
  - n.edges[0].t; // = 1
  - m.edges[0].t; // = 0
  - n.edges[0].weight == m.edges[0].weight; // 1
  - n.edges[0].dest == m; // 1
- All variants: ->, ->[weight], <-> (undirected), <->[weight]
- An expression “n -> m” evaluates to a reference to the node on the left (n), so these operators can be chained, and they are right associative.
  - n1 -> n2 -> n3 -> n4 -> ... // creates edges matching the visual structure of the expression
Features - Built-In Types - Graphs

Graphs are wrapper types that wrap a list of nodes of matching type.

- **Graphs contain a list of nodes**
  - `graph<int> g;`
  - `node<int>n; n.id = 1; n.val = 20;`
  - `g.add_node(n); // where n is of type node<int>`

- **Nodes in graphs can be indexed by their id**
  - `g[1]; // = n`

- **Node list of a graph can be accessed.**
  - `g.nodes[0] == g[1]; // 1`

- **Graphs have built-in functions that enable easier node creation**
  - `g.add(0, 1); // creates a node with id = 0, val = 1 and adds it to g`
  - `g.contains(0); // 1 if g contains a node with id = 0`
  - `g.contains_node(n); // 1 if g contains node n`
Misc. Features

- Parser supports chaining of accesses/indexes
  - `g.nodes[0].edges[2].dest.val; // valid expression`

- "Universal" print function
  - `print()` can take one argument of any primitive type, and by extension can print any field in Graphene.
  - Can also be passed a node as its argument, converts to calls to print for each field (size of edgelist)

- Improved variable declarations
  - `node<int> n, o, p, q, r, s, t, u, v, ...;`
  - `or`
  - `int i = 0;`
Architectural Design

- Source code (.gph) is scanned, parsed, semantically check, and translated to LLVM IR, which is then linked with a C library to produce the final executable.
- Structs are not actually supported, Graphene provides the illusion of structs/objects for its built-in types.
  - l.push_back(0);
  - parser outputs: push_back(l, 0)
C Library

- The C library is called from codegen to abstract some of the graph logic away from OCaml.
- Void pointers are sent between the files and casted accordingly in codegen where we have the types from the sast.

```c
struct list
{
    int size;
    struct list_element *head;
};

struct edge
{
    void *weight;
    struct node *dest;
    int t;
};

struct node
{
    int id;
    void *val;
    struct list *edges;
};

struct graph
{
    struct list *nodes;
    struct node *root;
};
```
Future Work

- Kill memory leaks
- Structs
- Support editing of graphs/nodes
- null
- break;
- continue;
- foreach
Demo Code

Graph:

Stacks:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Id * Id + Id$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>Id * Id$</td>
<td>Shift, goto 3</td>
</tr>
<tr>
<td>0 Id</td>
<td>Id + Id$</td>
<td>Shift, goto 1</td>
</tr>
<tr>
<td>0 Id</td>
<td>+ Id$</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 Id</td>
<td>+ Id$</td>
<td>Reduce 3</td>
</tr>
<tr>
<td>0</td>
<td>+ Id$</td>
<td>Shift, goto 4</td>
</tr>
</tbody>
</table>