KGL

Knowledge Graph Language

A domain-specific graphing language that supports multiple user-defined relationships between nodes.

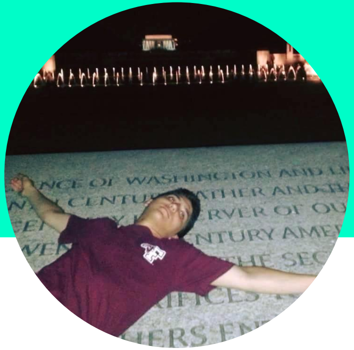
The Team

Ruoxin Jiang

System Architect

Cheng Huang

Language Guru



Nicholas Mariconda

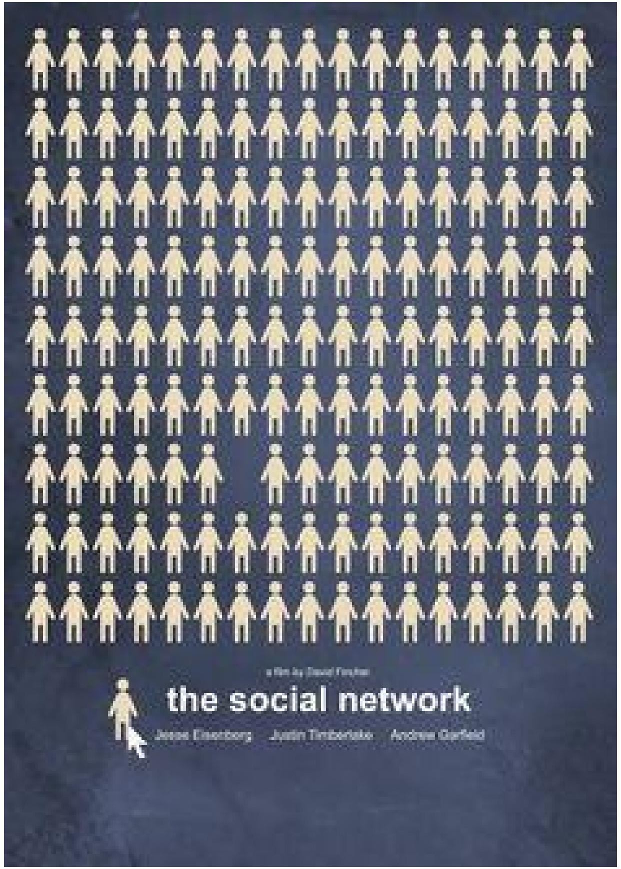
Tester

Bingyan Hu

Project Manager

Problem

- Complex web of relationships
- Hard to represent by graphs with traditional languages
- Limited by single, fixed relationship between nodes



Solution

Multi-relationship between nodes

Highly customizable attributes attached to the node

User-friendly built-in containers

Easily create and manipulate complex graph

Traverse and query graph

Language Features

Type

List & Dict

Node

- attributes (dict)
- outNeighbors
- inNeighbors

Graph

- nodes
- edges

Operator

[ ]

- list
- dict
- graph[name]
- node[attr]

+/+=

-/-=

- graph -= graph
- dict
- list

edge

- --()-->

In:

- enhanced for loops
- checks if an element exists

Built-in Functions

Print

Node-related

- getOutNeighbors()
- getInNeighbors()

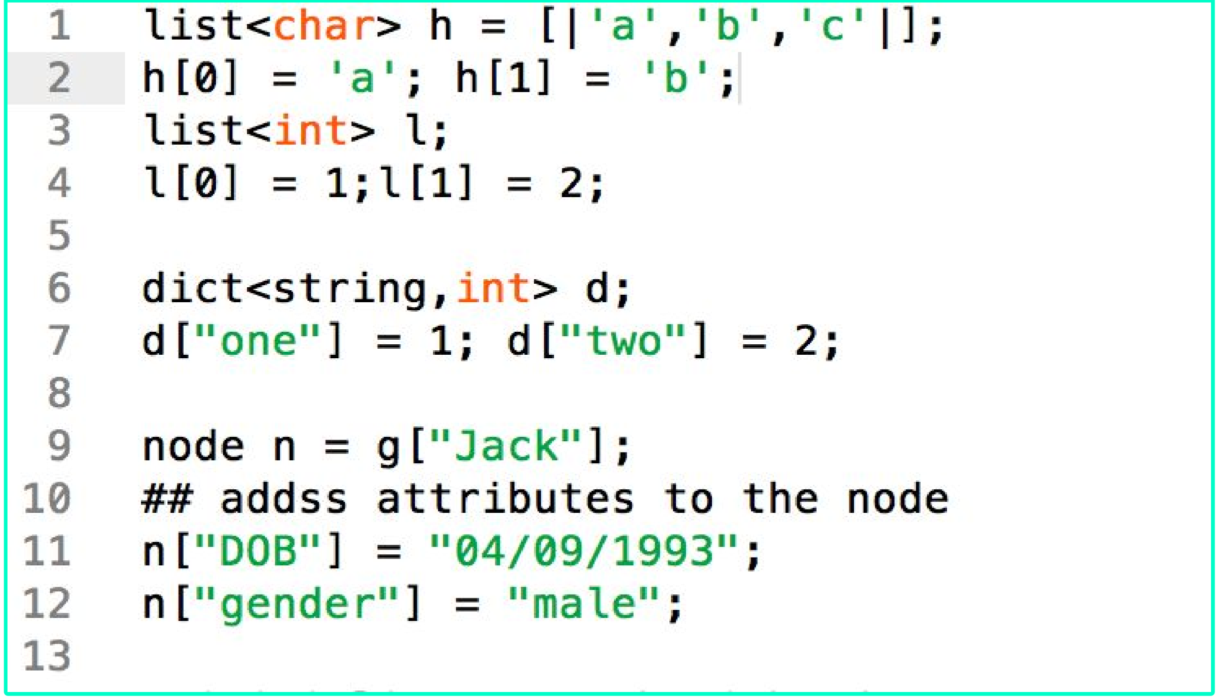
Graph-related

- getNodes()
- getLabels()

Language Features

Operator

[ ]	list dict graph[name] node[attr]
+/+= -/-=	graph dict list
edge	--()-->
In:	enhanced for loops checks if an element exists

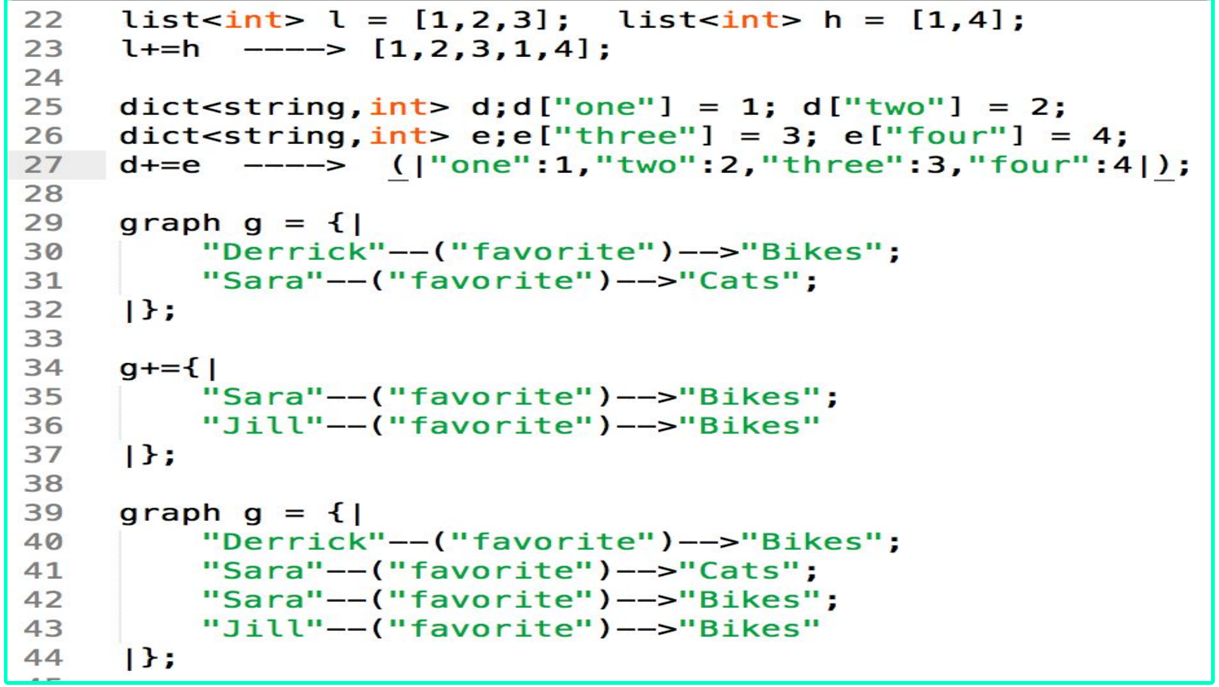


```
list<char> h = [|'a','b','c'|];
h[0] = 'a'; h[1] = 'b';
list<int> l;
l[0] = 1;l[1] = 2;

dict<string,int> d;
d["one"] = 1; d["two"] = 2;

node n = g["Jack"];
## addss attributes to the node
n["DOB"] = "04/09/1993";
n["gender"] = "male";

```

Language Features

Operator

[ ]	- list - dict - graph[name] - node[attr]
+/+= -/-=	- graph - dict - list
edge	- --()-->
In:	- enhanced for loops - checks if an element exists



```
list<int> l = [1,2,3];  list<int> h = [1,4];
l+=h  ----> [1,2,3,1,4];

dict<string,int> d;d["one"] = 1; d["two"] = 2;
dict<string,int> e;e["three"] = 3; e["four"] = 4;
d+=e  ---->  (|"one":1,"two":2,"three":3,"four":4|);

graph g = {|
        "Derrick"--("favorite")-->"Bikes";
        "Sara"--("favorite")-->"Cats";
|};

g+={|
        "Sara"--("favorite")-->"Bikes";
        "Jill"--("favorite")-->"Bikes"
|};

graph g = {|
        "Derrick"--("favorite")-->"Bikes";
        "Sara"--("favorite")-->"Cats";
        "Sara"--("favorite")-->"Bikes";
        "Jill"--("favorite")-->"Bikes"
|};
```

Language Features

Operator

[ ]

- list
- dict
- graph[name]
- node[attr]

+/+=
-/-=

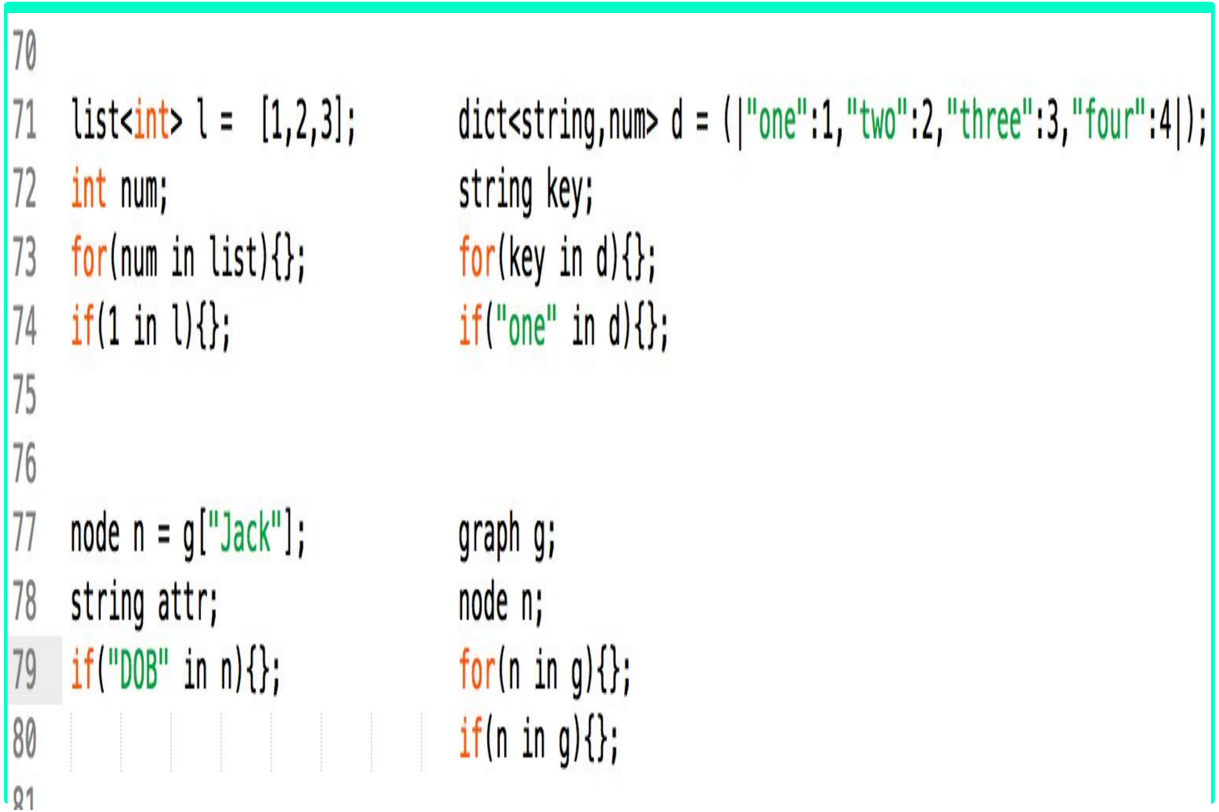
- graph
- dict
- list

edge

- --()-->

In:

- enhanced for loops
- checks if an element exists



```
list<int> l =  [1,2,3];        dict<string,num> d = (|"one":1,"two":2,"three":3,"four":4|);
int num;                       string key;
for(num in list){};            for(key in d){};
if(1 in l){};                  if("one" in d){};


node n = g["Jack"];            graph g;
string attr;                   node n;
if("DOB" in n){};              for(n in g){};
                               if(n in g){};
```

Language Features

Built-in Functions

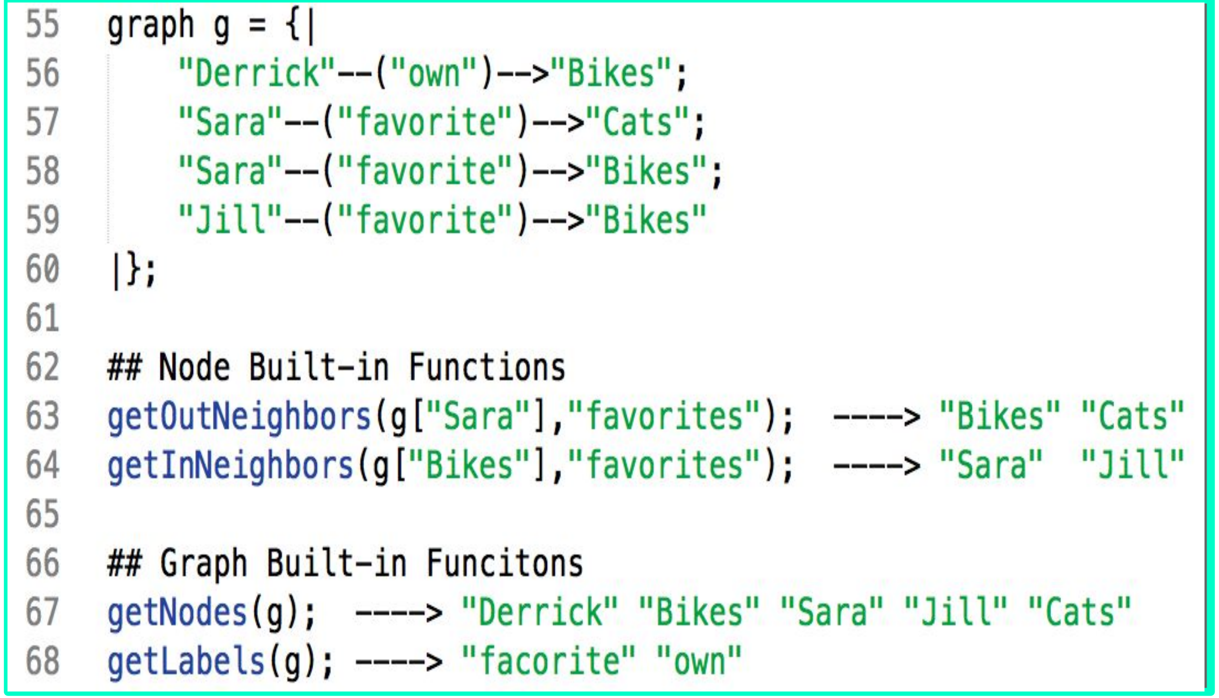
Print

Node-related

- getOutNeighbors()
- getInNeighbors()

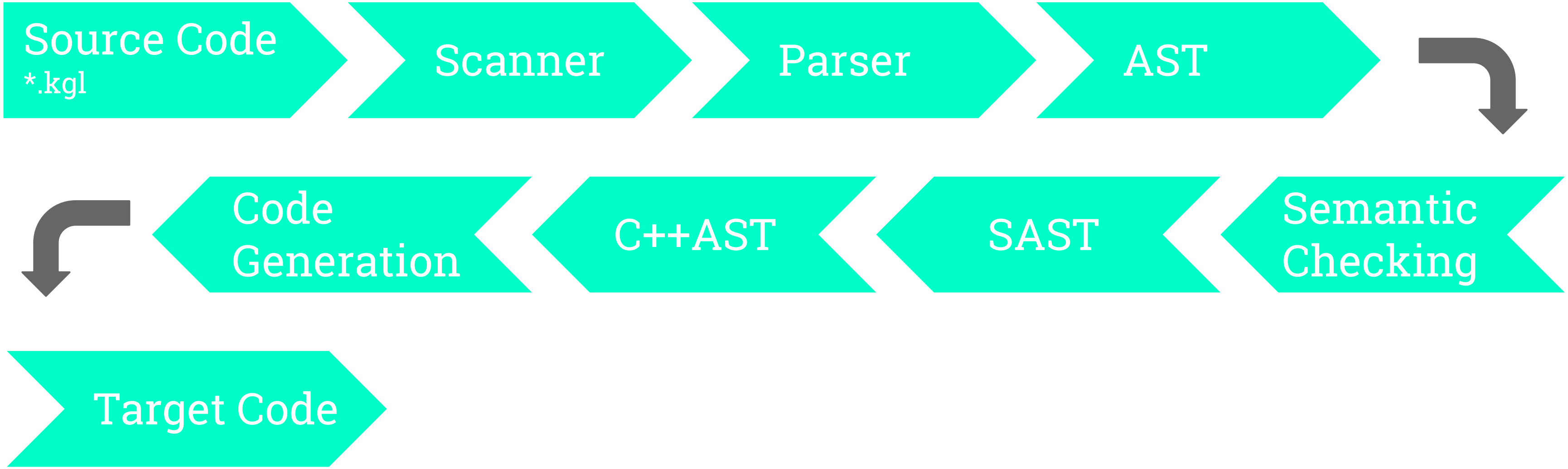
Graph-related

- getNodes()
- getLabels()



```
graph g = {|
    "Derrick"--("own")-->"Bikes";
    "Sara"--("favorite")-->"Cats";
    "Sara"--("favorite")-->"Bikes";
    "Jill"--("favorite")-->"Bikes"
|};

## Node Built-in Functions
getOutNeighbors(g["Sara"],"favorites");  ----> "Bikes" "Cats"
getInNeighbors(g["Bikes"],"favorites");  ----> "Sara"  "Jill"

## Graph Built-in Funcitons
getNodes(g);  ----> "Derrick" "Bikes" "Sara" "Jill" "Cats"
getLabels(g); ----> "facorite" "own"
```

Architecture



Source Code
*.kgl

Scanner

Parser

AST

Semantic Checking

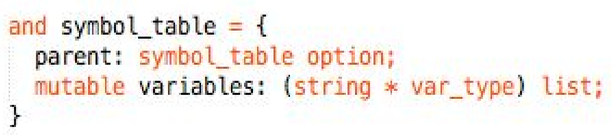
SAST

C++AST

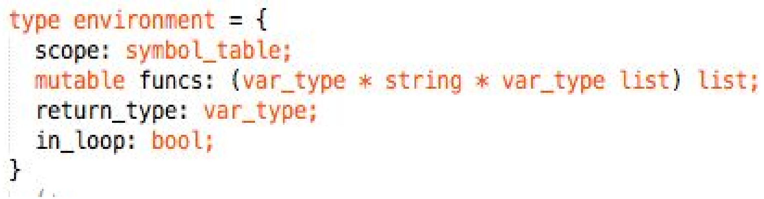
Code Generation

Target Code

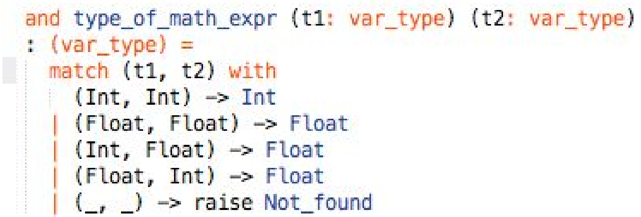
Semantic Checking



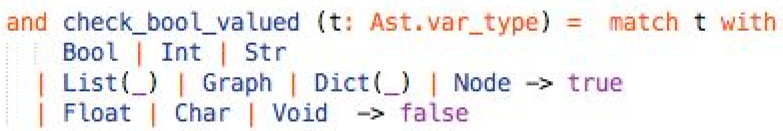
```
and symbol_table = {
  parent: symbol_table option;
  mutable variables: (string * var_type) list;
}
```



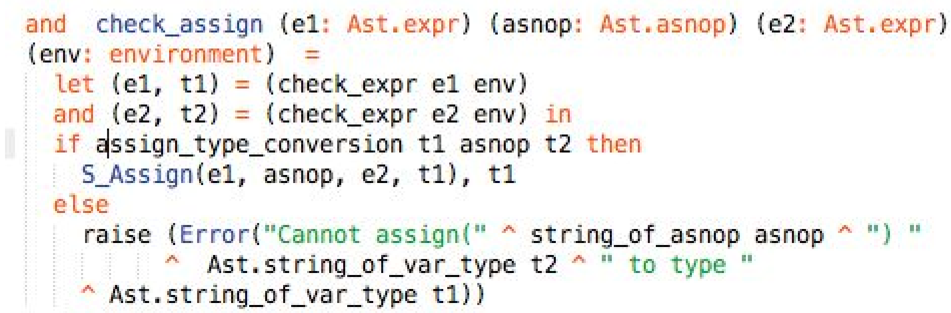
```
type environment = {
  scope: symbol_table;
  mutable funcs: (var_type * string * var_type list) list;
  return_type: var_type;
  in_loop: bool;
}
```



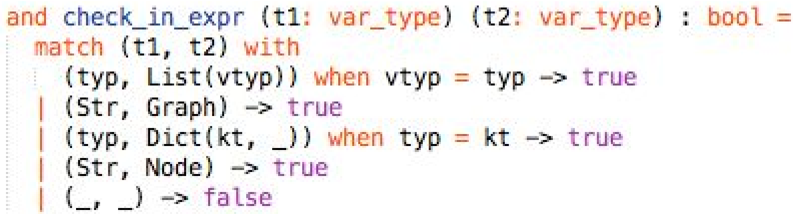
```
and type_of_math_expr (t1: var_type) (t2: var_type)
: (var_type) =
  match (t1, t2) with
    (Int, Int) -> Int
  | (Float, Float) -> Float
  | (Int, Float) -> Float
  | (Float, Int) -> Float
  | (_, _) -> raise Not_found
```



```
and check_bool_valued (t: Ast.var_type) =  match t with
    Bool | Int | Str
  | List(_) | Graph | Dict(_) | Node -> true
  | Float | Char | Void  -> false
```

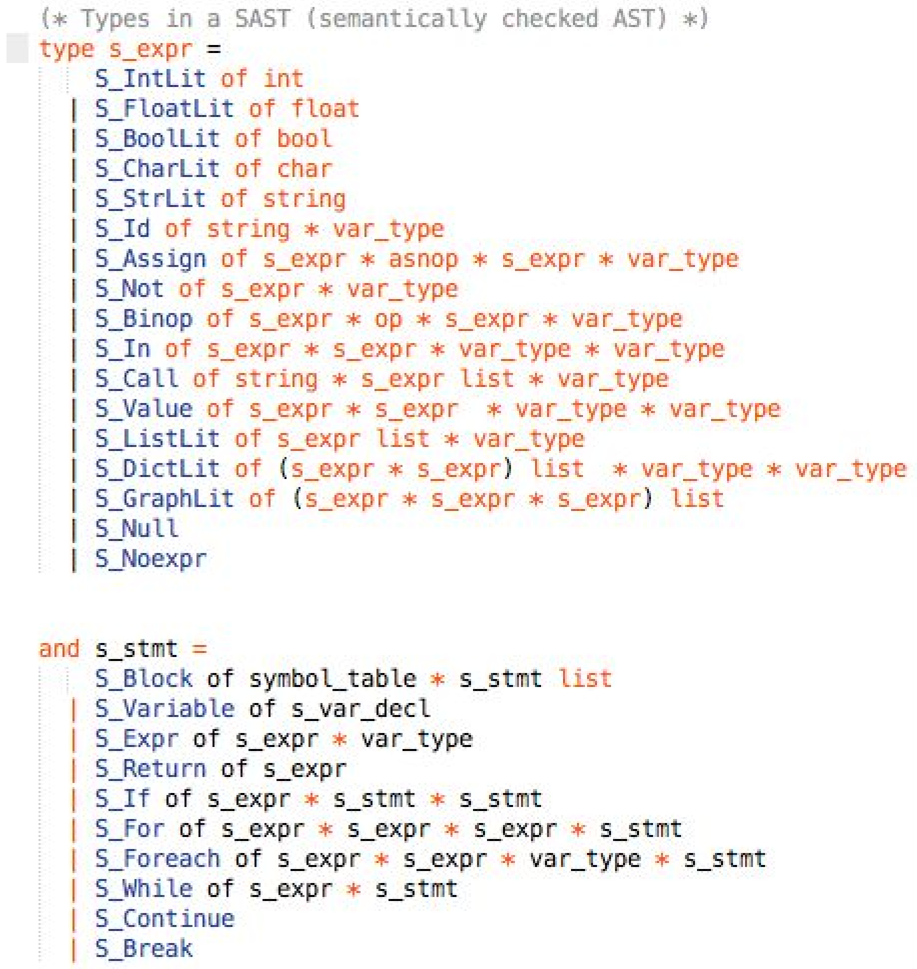


```
and  check_assign (e1: Ast.expr) (asnop: Ast.asnop) (e2: Ast.expr)
(env: environment)  =
  let (e1, t1) = (check_expr e1 env)
  and (e2, t2) = (check_expr e2 env) in
  if assign_type_conversion t1 asnop t2 then
    S_Assign(e1, asnop, e2, t1), t1
  else
    raise (Error("Cannot assign(" ^ string_of_asnop asnop ^ ") "
            ^  Ast.string_of_var_type t2 ^ " to type "
      ^ Ast.string_of_var_type t1))
```

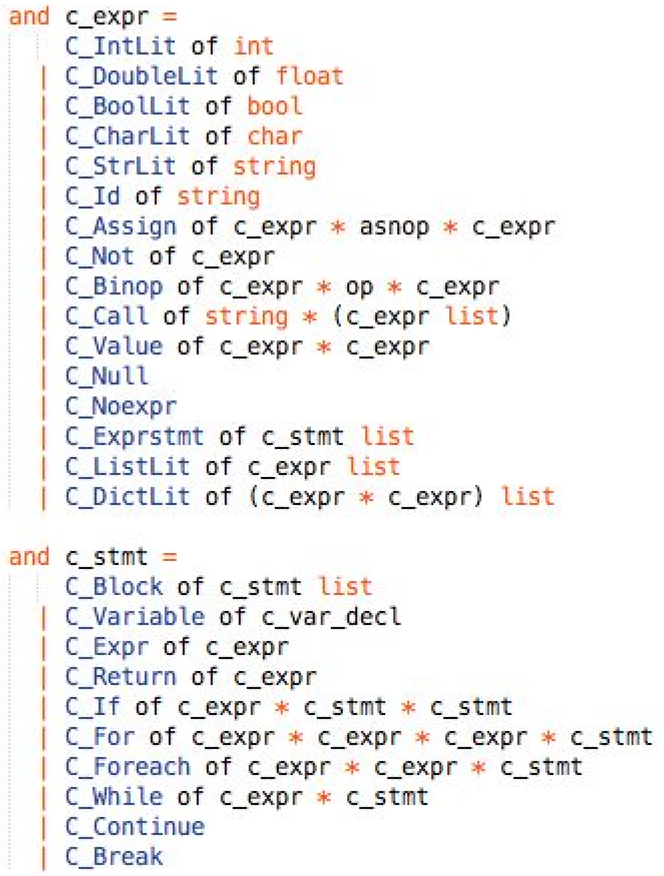


```
and check_in_expr (t1: var_type) (t2: var_type) : bool =
  match (t1, t2) with
    (typ, List(vtyp)) when vtyp = typ -> true
  | (Str, Graph) -> true
  | (typ, Dict(kt, _)) when typ = kt -> true
  | (Str, Node) -> true
  | (_, _) -> false
```

SAST -> C++ AST

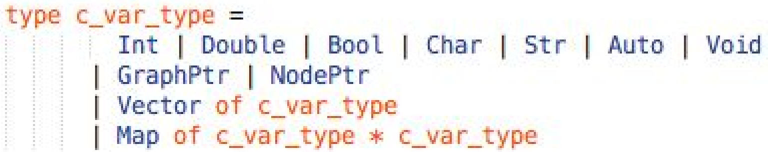


```
(* Types in a SAST (semantically checked AST) *)
type s_expr =
    S_IntLit of int
  | S_FloatLit of float
  | S_BoolLit of bool
  | S_CharLit of char
  | S_StrLit of string
  | S_Id of string * var_type
  | S_Assign of s_expr * asnop * s_expr * var_type
  | S_Not of s_expr * var_type
  | S_Binop of s_expr * op * s_expr * var_type
  | S_In of s_expr * s_expr * var_type * var_type
  | S_Call of string * s_expr list * var_type
  | S_Value of s_expr * s_expr  * var_type * var_type
  | S_ListLit of s_expr list * var_type
  | S_DictLit of (s_expr * s_expr) list  * var_type * var_type
  | S_GraphLit of (s_expr * s_expr * s_expr) list
  | S_Null
  | S_Noexpr

and s_stmt =
    S_Block of symbol_table * s_stmt list
  | S_Variable of s_var_decl
  | S_Expr of s_expr * var_type
  | S_Return of s_expr
  | S_If of s_expr * s_stmt * s_stmt
  | S_For of s_expr * s_expr * s_expr * s_stmt
  | S_Foreach of s_expr * s_expr * var_type * s_stmt
  | S_While of s_expr * s_stmt
  | S_Continue
  | S_Break
```



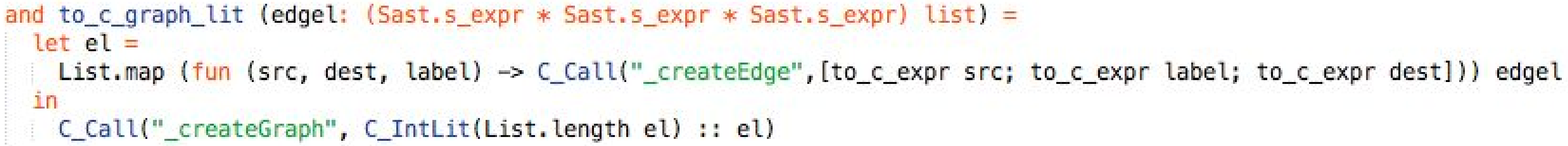
```
and c_expr =
    C_IntLit of int
  | C_DoubleLit of float
  | C_BoolLit of bool
  | C_CharLit of char
  | C_StrLit of string
  | C_Id of string
  | C_Assign of c_expr * asnop * c_expr
  | C_Not of c_expr
  | C_Binop of c_expr * op * c_expr
  | C_Call of string * (c_expr list)
  | C_Value of c_expr * c_expr
  | C_Null
  | C_Noexpr
  | C_Exprstmt of c_stmt list
  | C_ListLit of c_expr list
  | C_DictLit of (c_expr * c_expr) list

and c_stmt =
    C_Block of c_stmt list
  | C_Variable of c_var_decl
  | C_Expr of c_expr
  | C_Return of c_expr
  | C_If of c_expr * c_stmt * c_stmt
  | C_For of c_expr * c_expr * c_expr * c_stmt
  | C_Foreach of c_expr * c_expr * c_stmt
  | C_While of c_expr * c_stmt
  | C_Continue
  | C_Break
```

C++ AST



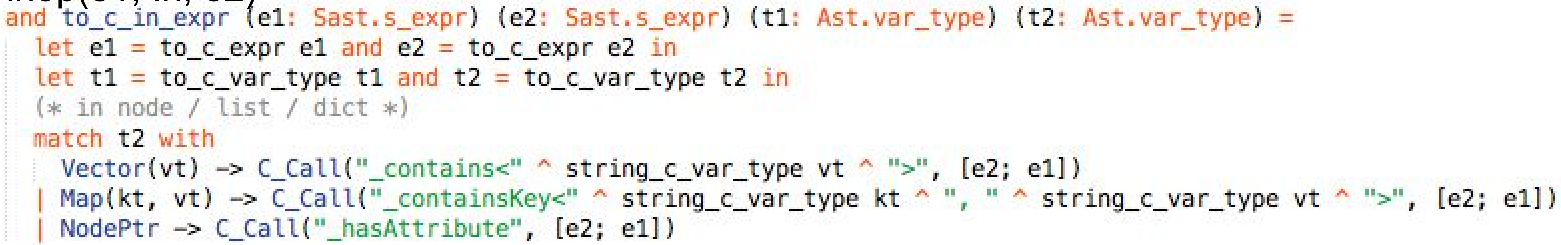
```
type c_var_type =
          Int | Double | Bool | Char | Str | Auto | Void
        | GraphPtr | NodePtr
        | Vector of c_var_type
        | Map of c_var_type * c_var_type
```

S_GraphLit



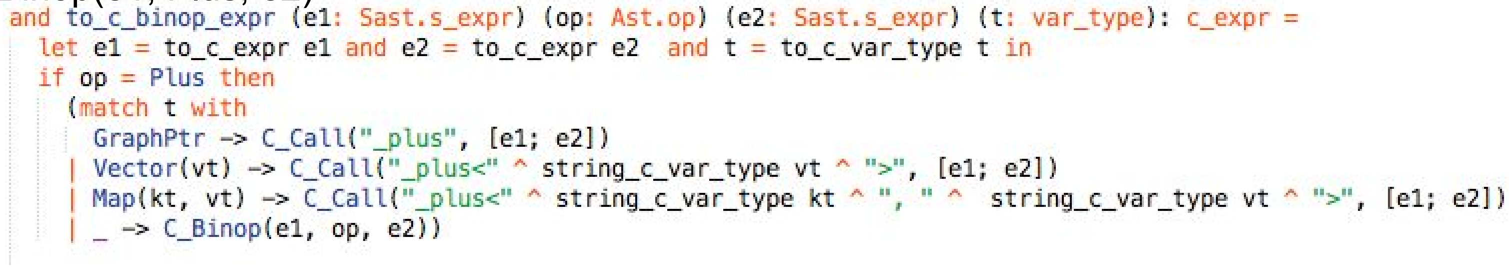
```
and to_c_graph_lit (edgel: (Sast.s_expr * Sast.s_expr * Sast.s_expr) list) =
  let el =
    List.map (fun (src, dest, label) -> C_Call("_createEdge",[to_c_expr src; to_c_expr label; to_c_expr dest])) edgel
  in
    C_Call("_createGraph", C_IntLit(List.length el) :: el)
```

S_Binop(e1, In, e2)



```
and to_c_in_expr (e1: Sast.s_expr) (e2: Sast.s_expr) (t1: Ast.var_type) (t2: Ast.var_type) =
  let e1 = to_c_expr e1 and e2 = to_c_expr e2 in
  let t1 = to_c_var_type t1 and t2 = to_c_var_type t2 in
  (* in node / list / dict *)
  match t2 with
    Vector(vt) -> C_Call("_contains<" ^ string_c_var_type vt ^ ">", [e2; e1])
  | Map(kt, vt) -> C_Call("_containsKey<" ^ string_c_var_type kt ^ ", " ^ string_c_var_type vt ^ ">", [e2; e1])
  | NodePtr -> C_Call("_hasAttribute", [e2; e1])
```

S_Binop(e1, Plus, e2)



```
and to_c_binop_expr (e1: Sast.s_expr) (op: Ast.op) (e2: Sast.s_expr) (t: var_type): c_expr =
  let e1 = to_c_expr e1 and e2 = to_c_expr e2  and t = to_c_var_type t in
  if op = Plus then
    (match t with
      GraphPtr -> C_Call("_plus", [e1; e2])
    | Vector(vt) -> C_Call("_plus<" ^ string_c_var_type vt ^ ">", [e1; e2])
    | Map(kt, vt) -> C_Call("_plus<" ^ string_c_var_type kt ^ ", " ^  string_c_var_type vt ^ ">", [e1; e2])
    | _ -> C_Binop(e1, op, e2))
```

Test Suite

Unit

Regression

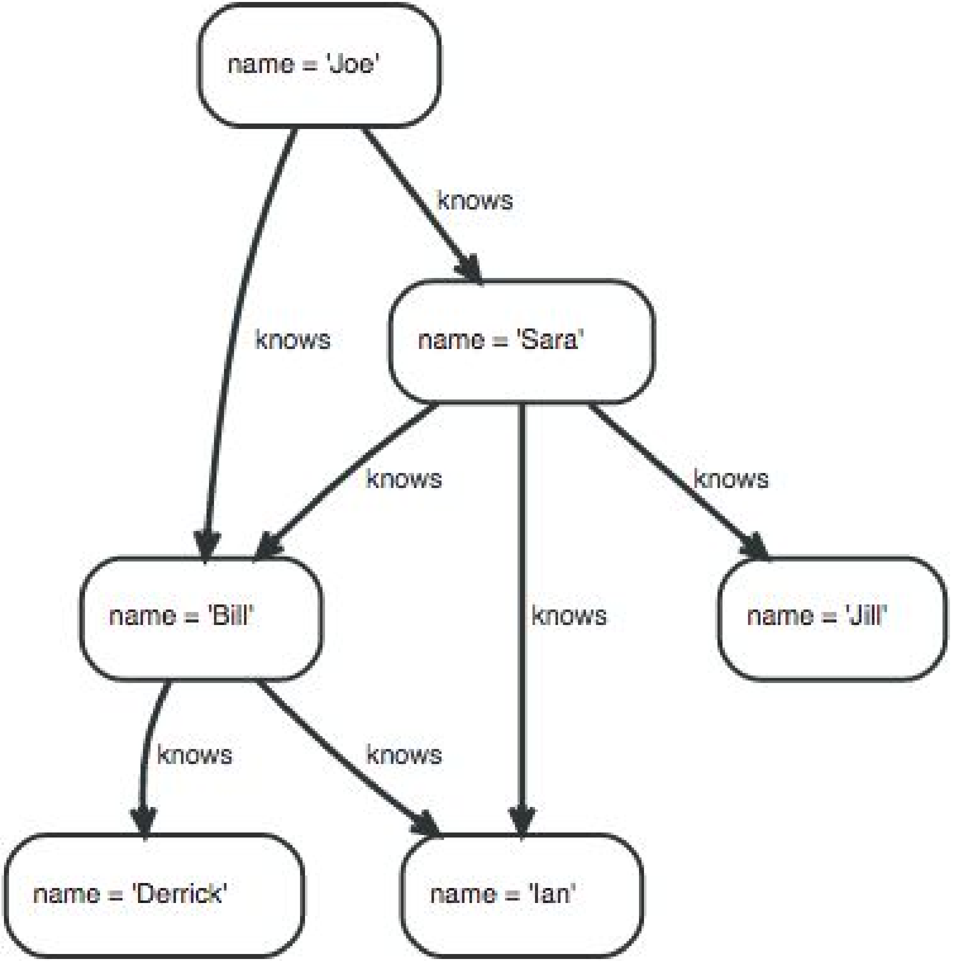
Integration

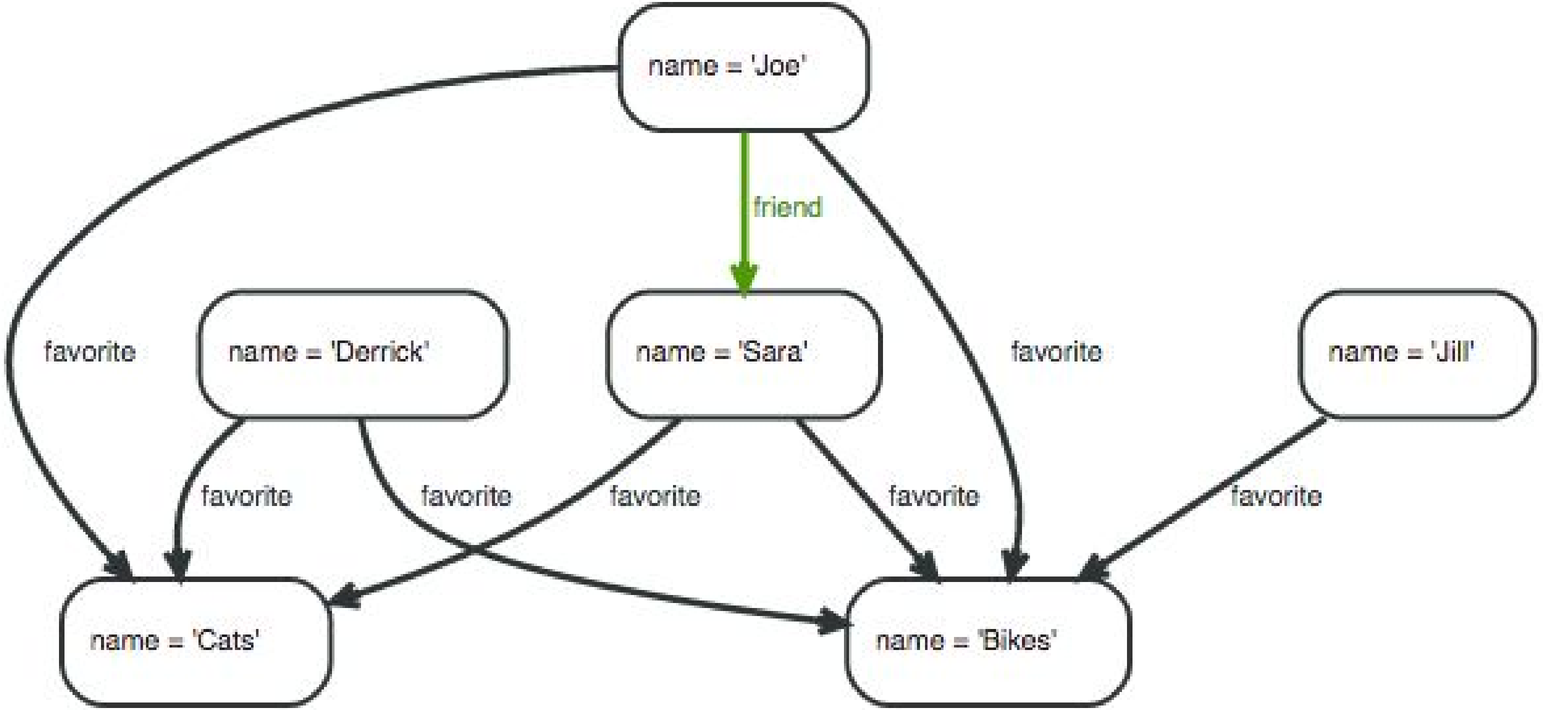
Test script recursively traverses test suite to run over 100 tests

Automatically generates output file for newly created tests

DEMO

Find your friend!


name = 'Joe'
knows
knows
name = 'Sara'
knows
knows
knows
name = 'Bill'
name = 'Jill'
knows
knows
name = 'Derrick'
name = 'Ian'


name = 'Joe'
friend
favorite
favorite
name = 'Derrick'
name = 'Sara'
name = 'Jill'
favorite
favorite
favorite
favorite
favorite
name = 'Cats'
name = 'Bikes'

Thank you!