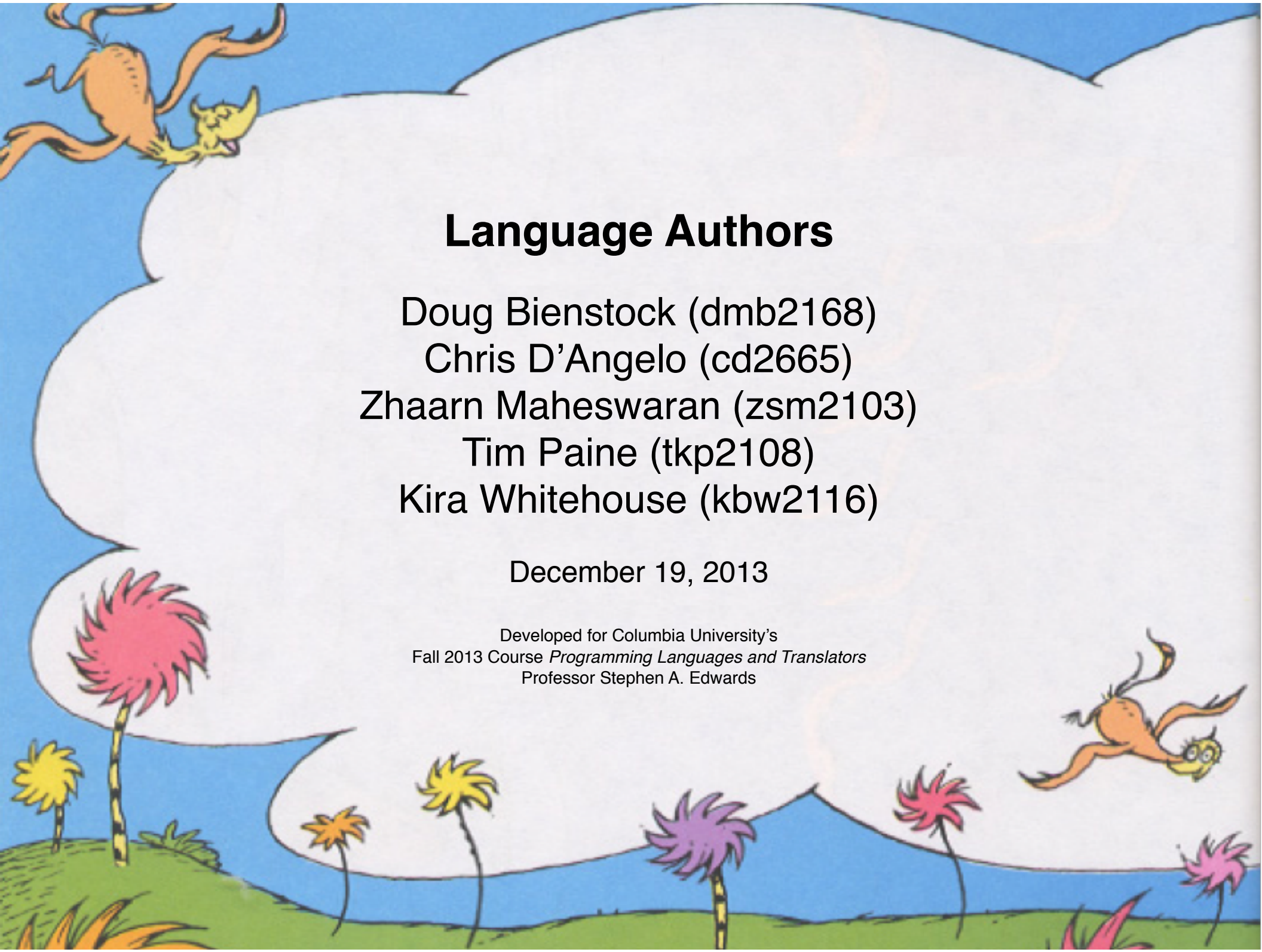

LORAX
The Language that Speaks for the Trees

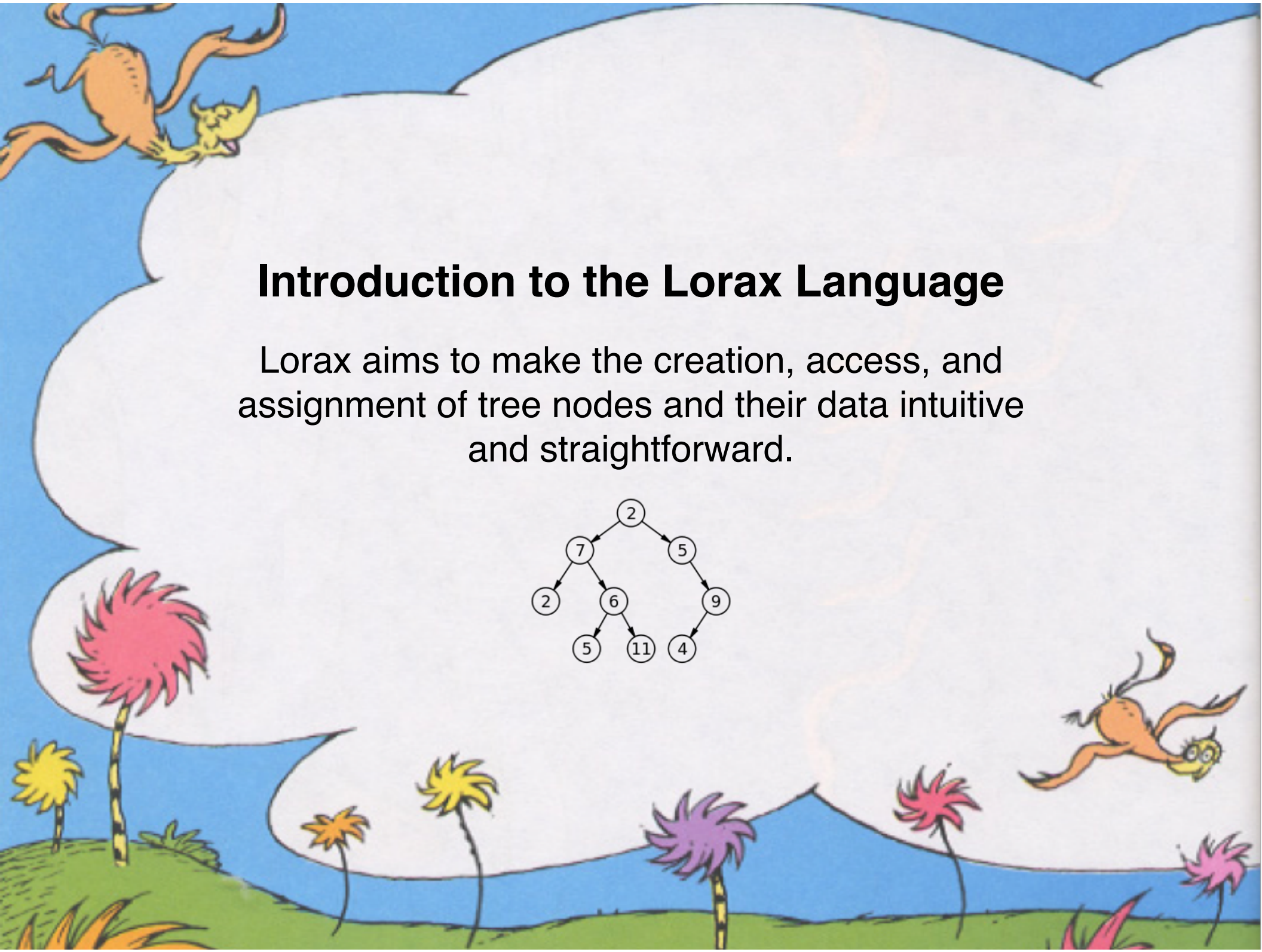


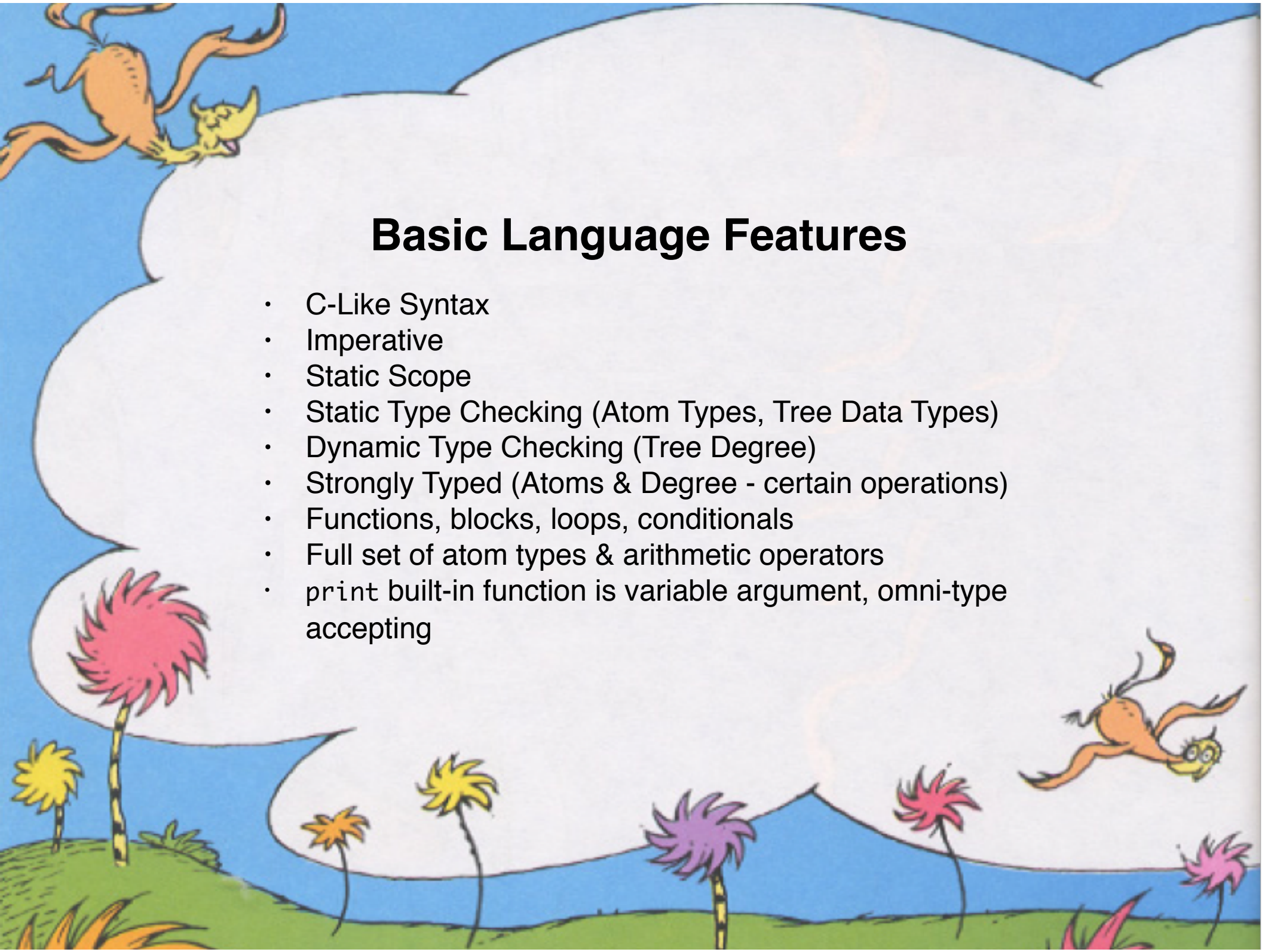
Language Authors

Doug Bienstock (dmb2168)
Chris D'Angelo (cd2665)
Zhaarn Maheswaran (zsm2103)
Tim Paine (tkp2108)
Kira Whitehouse (kbw2116)

December 19, 2013

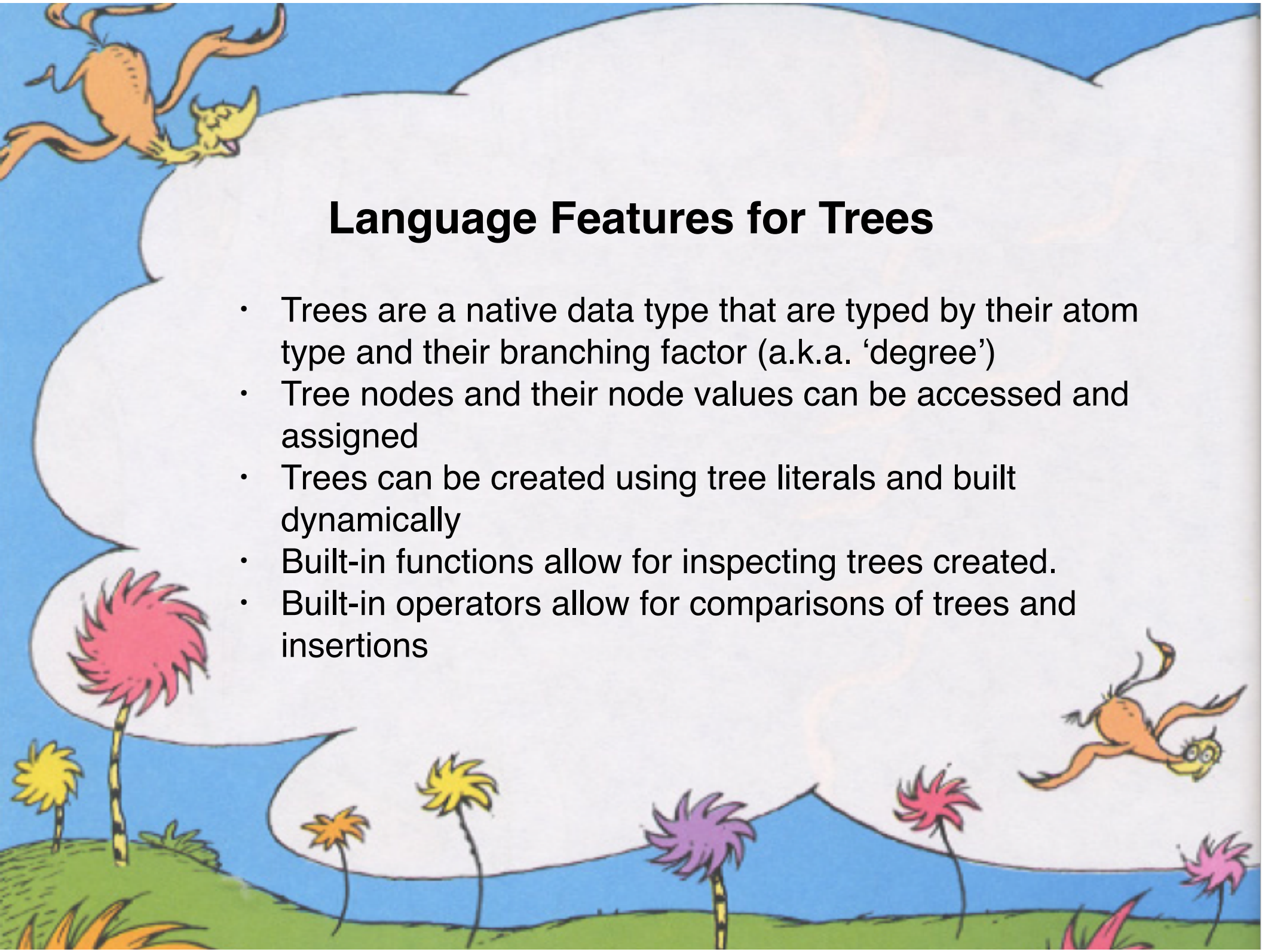
Developed for Columbia University's
Fall 2013 Course *Programming Languages and Translators*
Professor Stephen A. Edwards


Introduction to the Lorax Language
Lorax aims to make the creation, access, and assignment of tree nodes and their data intuitive and straightforward.
2
7
5
2
6
9
5
11
4



Basic Language Features

- C-Like Syntax
- Imperative
- Static Scope
- Static Type Checking (Atom Types, Tree Data Types)
- Dynamic Type Checking (Tree Degree)
- Strongly Typed (Atoms & Degree - certain operations)
- Functions, blocks, loops, conditionals
- Full set of atom types & arithmetic operators
- `print` built-in function is variable argument, omni-type accepting



Language Features for Trees

- Trees are a native data type that are typed by their atom type and their branching factor (a.k.a. ‘degree’)
- Tree nodes and their node values can be accessed and assigned
- Trees can be created using tree literals and built dynamically
- Built-in functions allow for inspecting trees created.
- Built-in operators allow for comparisons of trees and insertions

Tutorial

Lorax in One Slide*

*not even close

```
/* simple.lrx */

tree <int>t(2); // tree with type int, degree 2

int add_hundred(int a, tree<int>t(2)) {
   print("before reassignment: t = ", t, "\n");
   return a + 100;
}

int main() {
   int i;
   t = 1[2, 3[4, 5]];
   t%1%0@ = add_hundred(t%1%0@, t);
   print("after assignment: t = ", t, "\n");
}
```

global declaration and auto-initialization

multiline comment

function declaration/ definition. Can be called recursively. Takes any kind/number of arguments. Can return an atom type.

can be called recursively

single-line comment

pass by value

pass by reference

normal arithmetic operations

type/degree checking

local atom variable. also available float, char, bool

tree access & assignment: access second child, then that child's first, then its data member

tree literal definition and assignment

string literal converted to tree of char type 1 degree, and printing in string representation

escape char

variable argument print accepting omni-types

Running a Lorax Program

```
> cat simple.lrx
/* Simple Example */

tree <int>t(2); // tree with type int, degree 2

int make_hundred(int a, tree<int>t(2)) {
  print("before reassignment: t = ", t, "\n");
  return a + 100;
}

int main() {
  int i;
  t = 1[2, 3[4, 5]];
  t%1%0@ = make_hundred(t%1%0@, t);
  print("after assignment: t = ", t, "\n");
}%
> ./lorax -b simple.lrx
> ./a.out
before reassignment: t = 1[2[null,null],3[4[null,null],
5[null,null]]]
after assignment: t = 1[2[null,null],3[104[null,null],
5[null,null]]]
```

more flags for debugging available

-b compiles lorax output to c using gcc and writes a.out by default or third ./lorax argument as filename

Tree Features: Kitchen Sink

```
int main()
{
   tree <int>t(2); // tree declaration
   string s; // syntactic sugar for tree <char>s(1)
   tree <char>s2(1);
   s = "hi"; // conversion of string literal to tree
   s2 = 'h'['i'];
   t = 1[2, 3]; // tree initialization

   // structure and value comparison
   print("1. ", t == 1[2, 3], "\n");

   // structure and value comparison
   print("2. ", s == s2, "\n");

   // tree node access and dereference data member
   print("3. ", t%0@, "\n");

   // get parent of first child and get it's value
   print("4. ", parent(t%0)@, "\n");
}
```

Terminal Output

```
> ./a.out
1. true
2. true
3. 2
4. 1
```

Tree Features: Kitchen Sink II

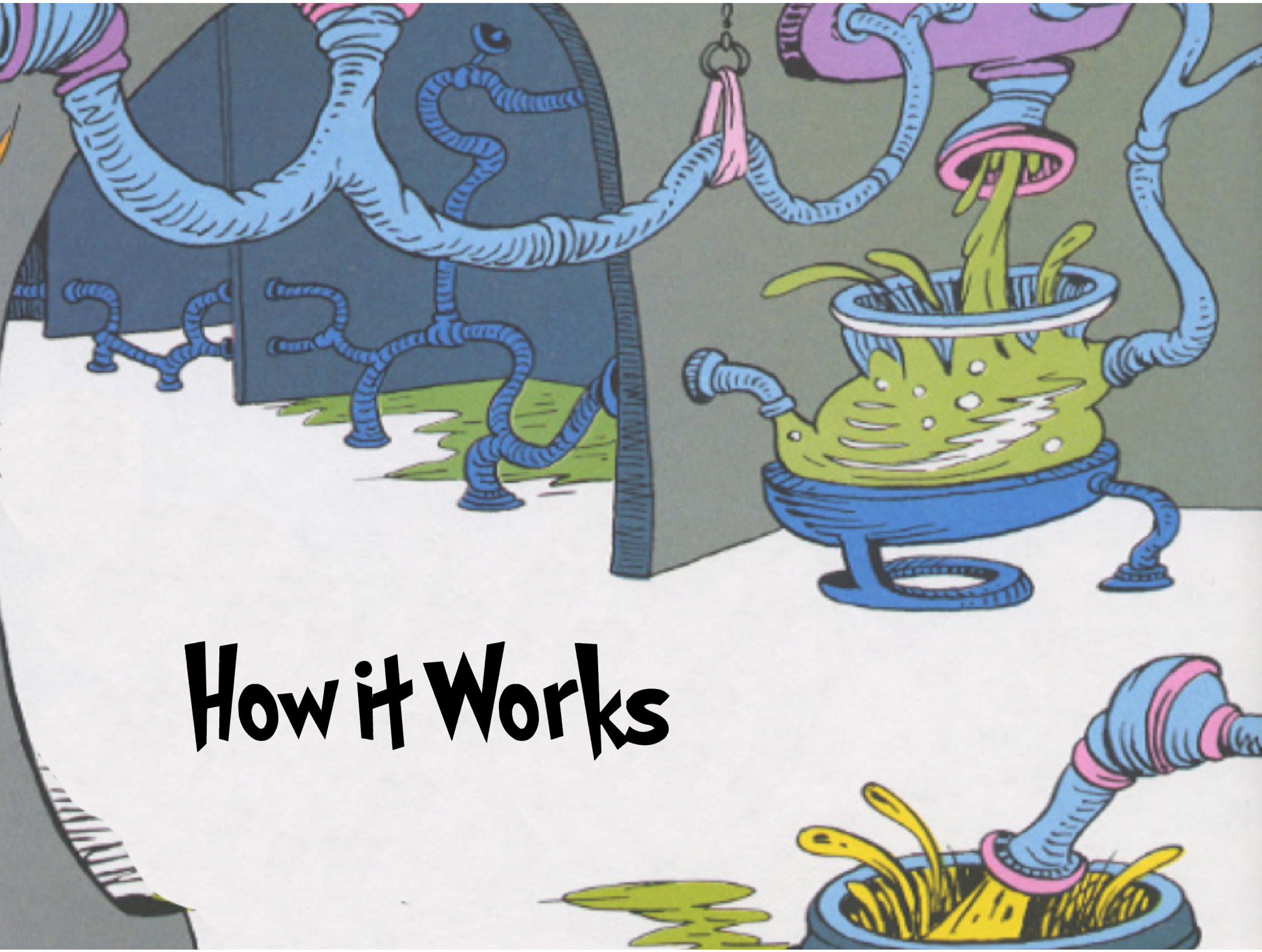
```
int main()
{
   tree <int>t(2);
   tree <float>t2(2);
   string s;

   t2 = 1.0[2.0, 3.0];
   t = 1[2, 3];
   t = t + 4[5, 6];

   print("after insertion to closest bfs null node = ",
   t, "\n");

   print("# of nodes comparison = ",
   'a'['b', 'c'['d', 'e']] > t2, "\n");

   // compare to null to see if child exists
   if (t2%0%0 == null) {
      s = "nope";
   } else {
      s = "yep";
   }

   print("Does t2 have a grandchild? ", s, "\n");
}
```

Terminal Output

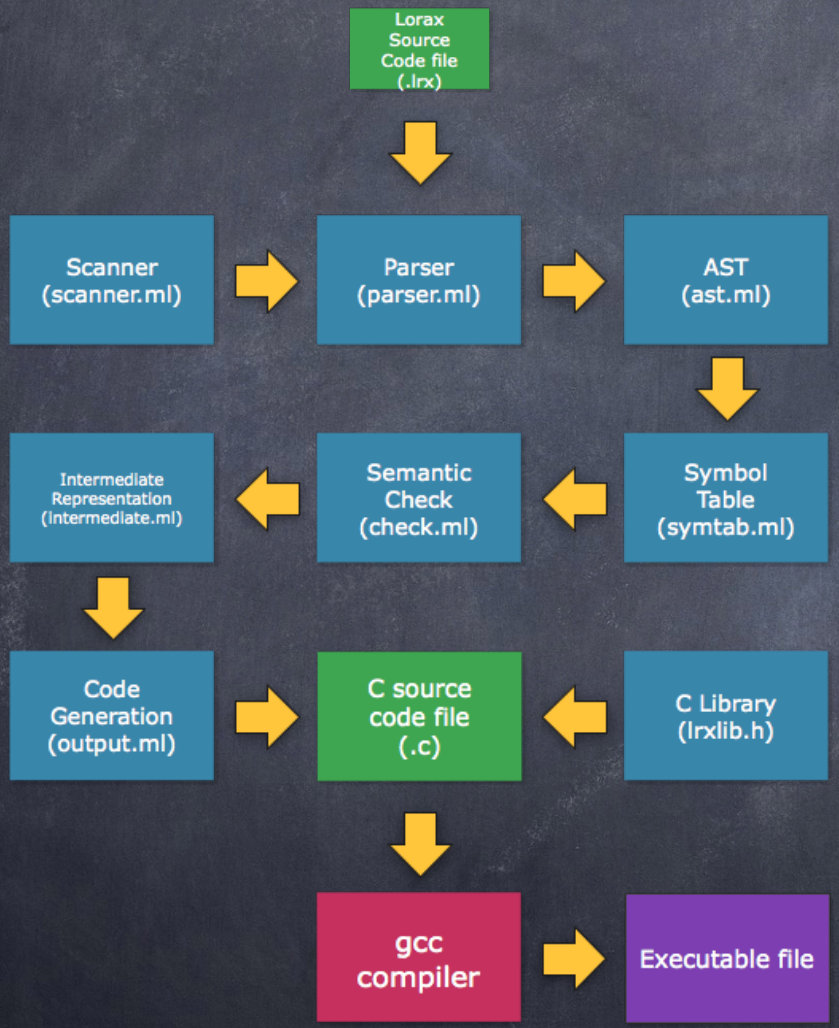
```
> ./a.out
after insertion to closest
bfs null node =
1[2[4[5[null,null],
6[null,null]],null],
3[null,null]]
# of nodes comparison =
true
Does t2 have a grandchild?
nope
```

Full Example: Depth First Search

```
bool dfs(tree <int>t(2), int val) {
  int child;
  bool match;
  match = false;
  if (t == null) { return false; }
  if (t@ == val) { return true; }
  for (child = 0; child < degree(t); child = child + 1) {
    if (t%child != null) {
      if(t%child@ == val) { return true; }
      else { match = dfs(t%child, val); }
    }
  }
  return match;
}
int main() {
  tree <int>t(2);
  t = 1[2, 3[4, 5]];
  if (dfs(t, 3)) { print("found it\n"); }
  else { print("its not there\n"); }
}
```

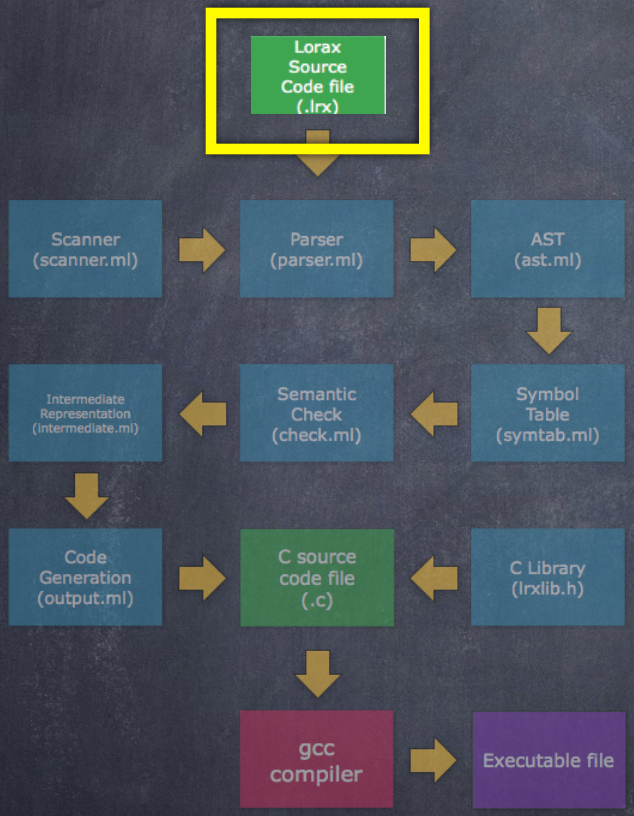

How it Works

How it Works: Overview



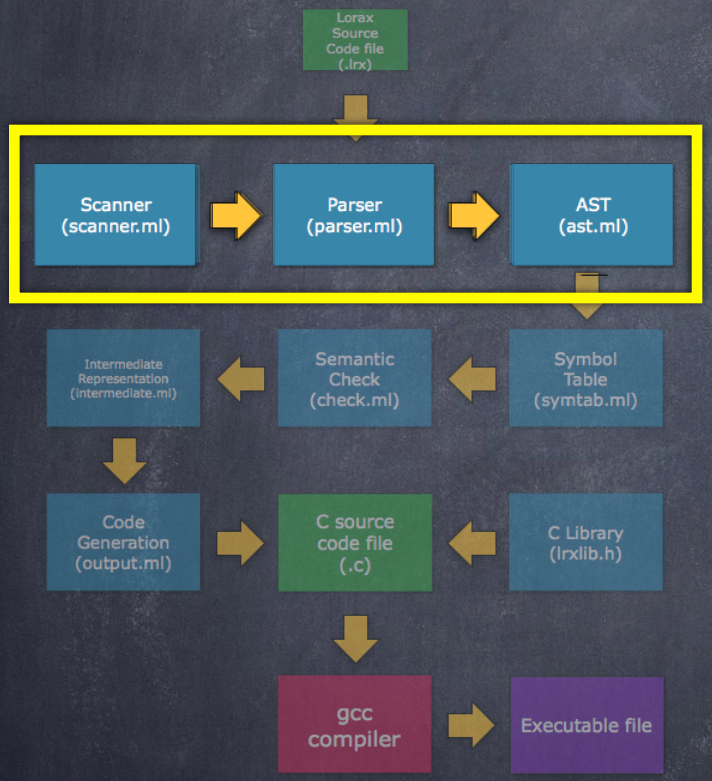
- **scanner.ml**: separates source code into tokens
- **parser.ml**: parses tokens into AST
- **symtab.ml**: builds symbol table for all identifiers
- **check.ml**: validates AST
- **intermediate.ml**: flattens all but function def/calls into a list of three address like code
- **output.ml**: converts intermediate types to c compatible syntax

How it Works: Source Code



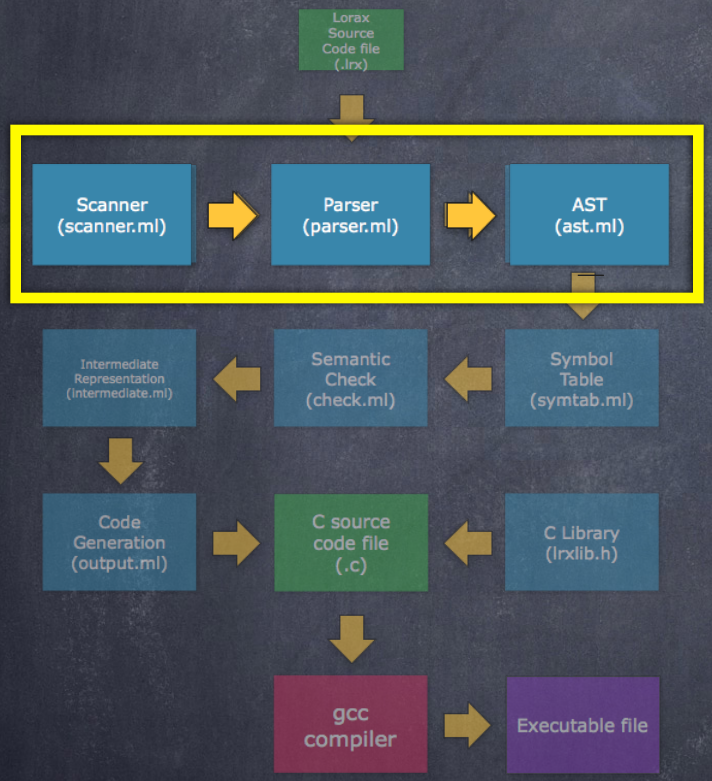

```
// lorax source code
int main()
{
  tree <int>t(2);
  t = 1[2, 3];
  print(t);
}
```

How it Works: Abstract Syntax Tree



- Typical Scanning/Parsing process. Source code is broken into tokens and parsed into an abstract syntax tree
- During Parsing identifiers are given a unique number in every scope { }

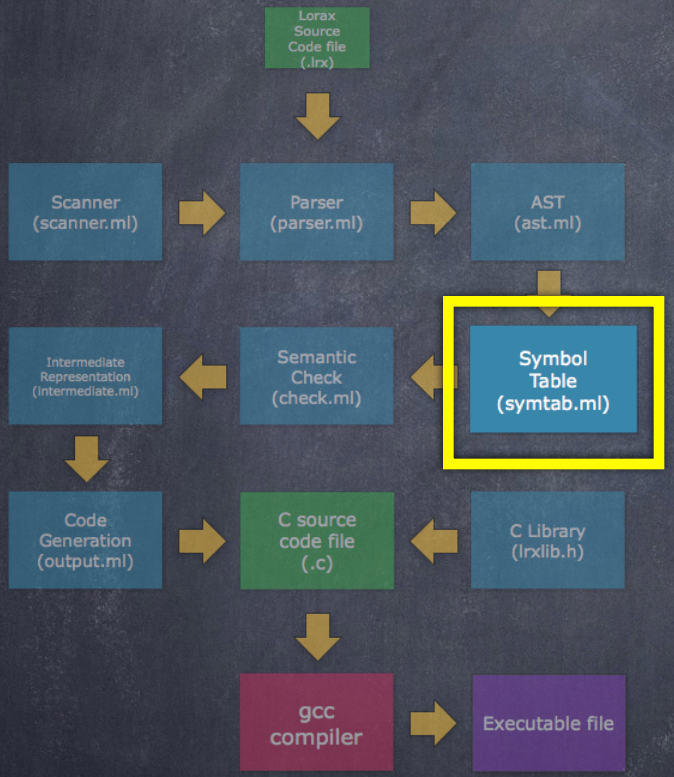
How it Works: Abstract Syntax Tree



Scanning & Parsing results in abstract syntax tree

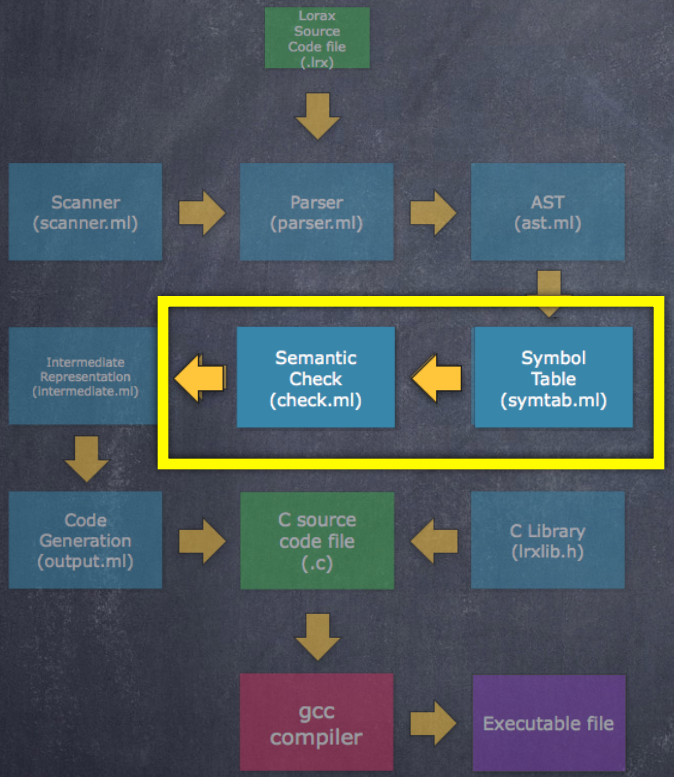
```
program: Ast.program =
  ([],
   [{Ast.fname = "main"; ret_type = Ast.Lrx_Atom Ast.Lrx_Int;
formals = [];
     fblock =
      {Ast.locals =
        [("t",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree =
Ast.Int_Literal 2})];
       statements =
        [Ast.Expr
          (Ast.Assign (Ast.Id "t",
            Ast.Tree (Ast.Int_Literal 1,
              [Ast.Int_Literal 2; Ast.Int_Literal 3])));
         Ast.Expr (Ast.Call ("print", [Ast.Id "t"]))];
       block_id = 1}}])
```

How it Works: Symbol Table



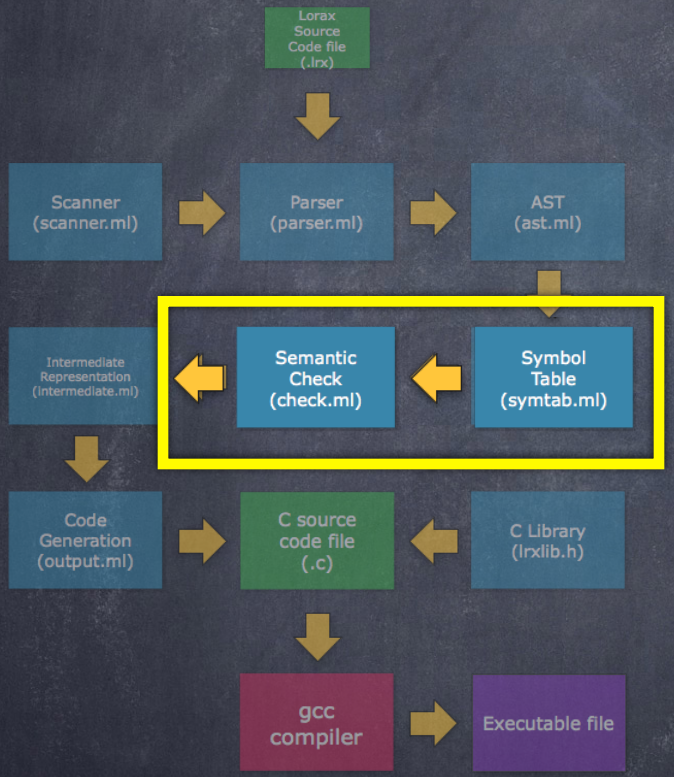
- Symbol Table creates a string map where identifiers appended with their scope number is the key. If the key exists twice an error is thrown.

How it Works: Checked AST



- Semantic Analysis performs type checking and performs validation on semantics not possible within the parsing process.
- **Unique to Lorax**
  - Tree Literals (e.g. 1[2, 3]) are verified for type consistency and degree consistency
  - Tree operations are verified for atom type consistency. Degree consistency is left for runtime checking.

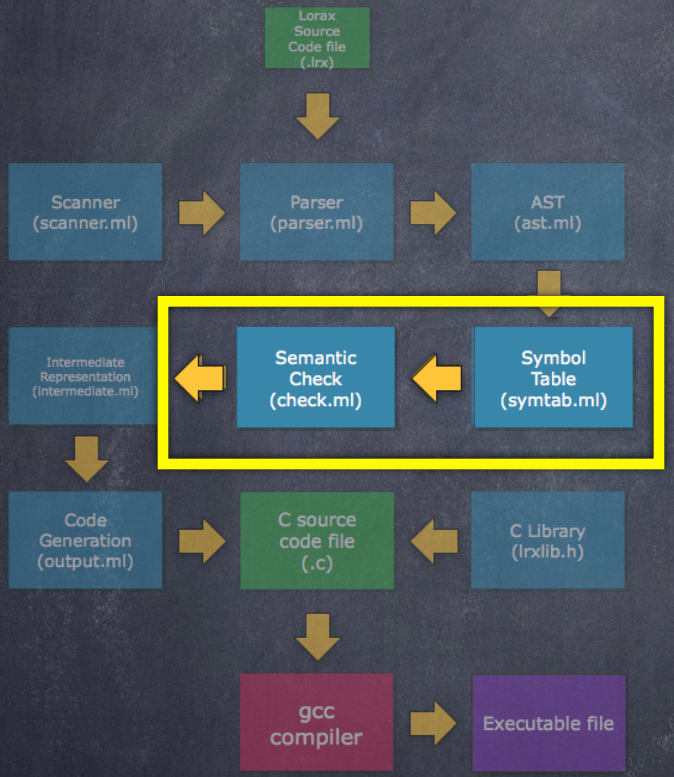
How it Works: Checked AST



Code Snippet Critical to Tree Literal Validation

```
and check_tree_literal_is_valid (d:int) (t:var_type) (el:expr list) env =
     match el with
        [] -> []
      | head :: tail ->
        let checked_expr = check_expr head env in
        match checked_expr with
           C_Tree(tree_type, tree_degree, child_e, child_el) ->
           if (tree_degree = d || tree_degree = 0) && tree_type = t then
             C_Tree(tree_type, d, child_e, child_el) ::
check_tree_literal_is_valid d t tail env
           else raise (Failure ("Tree type is not consistent: expected <" ^
string_of_var_type t ^ ">(" ^ string_of_int d ^ ") but received <" ^
string_of_var_type tree_type ^ ">(" ^ string_of_int tree_degree ^ ")"))
         | _ ->
           let child_type = (type_of_expr checked_expr) in
           if child_type = t then
             checked_expr :: check_tree_literal_is_valid d t tail env
           else raise (Failure ("Tree literal type is not consistent: expected
<" ^ string_of_var_type t ^ "> but received <" ^ string_of_var_type child_type
^">"))

and check_tree_literal_root_is_valid (e:expr) (el: expr list) env =
  let checked_root = check_expr e env in
  let type_root = type_of_expr checked_root in
  match type_root with
     (Lrx_Atom(Lrx_Int) | Lrx_Atom(Lrx_Float) | Lrx_Atom(Lrx_Char) |
Lrx_Atom(Lrx_Bool)) ->
     let degree_root = List.length el in
     let checked_tree = check_tree_literal_is_valid degree_root type_root el
env in
     (type_root, degree_root, checked_root, checked_tree)
   | _ -> raise (Failure ("Tree root cannot be of non-atom type: " ^
string_of_var_type type_root))
```

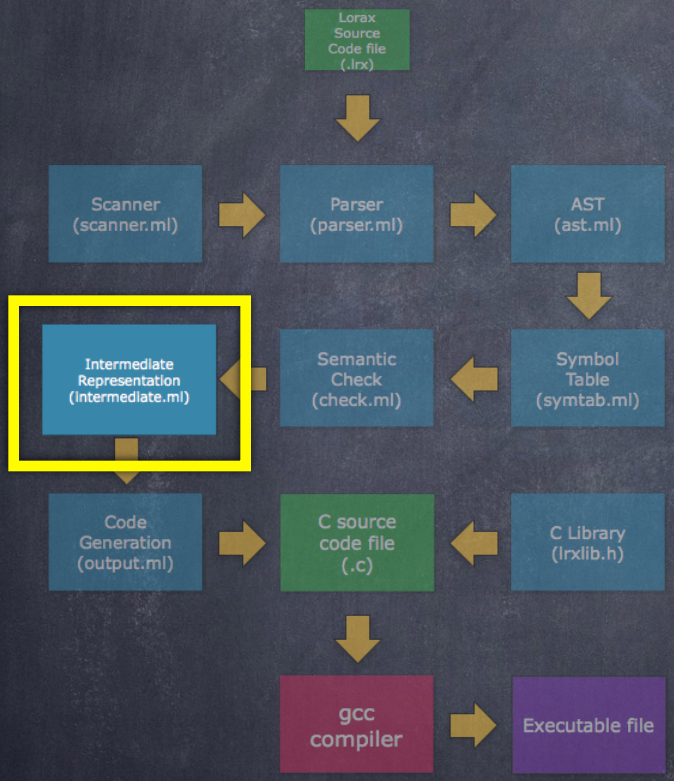
How it Works: Checked AST



Semantic Analysis results in *Checked* AST

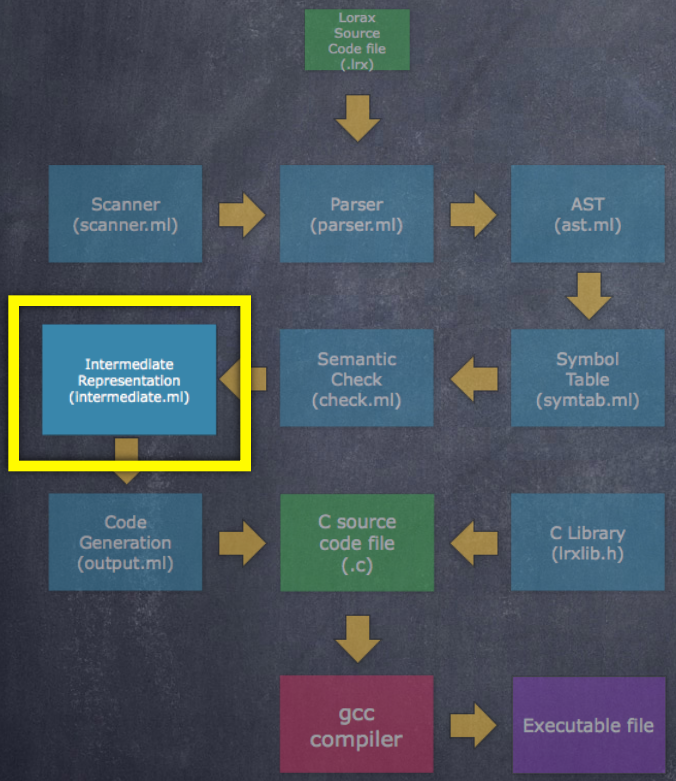
```
checked: Ast.scope_var_decl list * Check.c_func list =
  ([],
   [{Check.c_fname = "main"; c_ret_type = Ast.Lrx_Atom Ast.Lrx_Int;
     c_formals = [];
     c_fblock =
      {Check.c_locals =
        [("t",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          1)];
       c_statements =
        [Check.C_Expr
          (Check.C_Assign
            (Ast.Lrx_Tree
              {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
             Check.C_Id
              (Ast.Lrx_Tree
                {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
               "t", 1),
             Check.C_Tree (Ast.Lrx_Atom Ast.Lrx_Int, 2, Check.C_Int_Literal 1,
              [Check.C_Int_Literal 2; Check.C_Int_Literal 3])));
         Check.C_Expr
          (Check.C_Call
            (("print", Ast.Lrx_Atom Ast.Lrx_Int,
              [Ast.Lrx_Tree
                {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2}],
              0),
             [Check.C_Id
               (Ast.Lrx_Tree
                 {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
                "t", 1)]))];
       c_block_id = 1}}])
```

How it Works: Intermediate Representation



- Intermediate Representation phase converts the semantically checked AST into a flat list 3 address like code representation.
- We intentionally underutilized conveniences of c language. Example: all control flow is flattened into `gotos` instead of using c-language native looping. However, c function calls were utilized.
- Often this Checked AST to IR conversion requires specifically tailored algorithms that builds up a list of necessary statements from a single AST instruction.

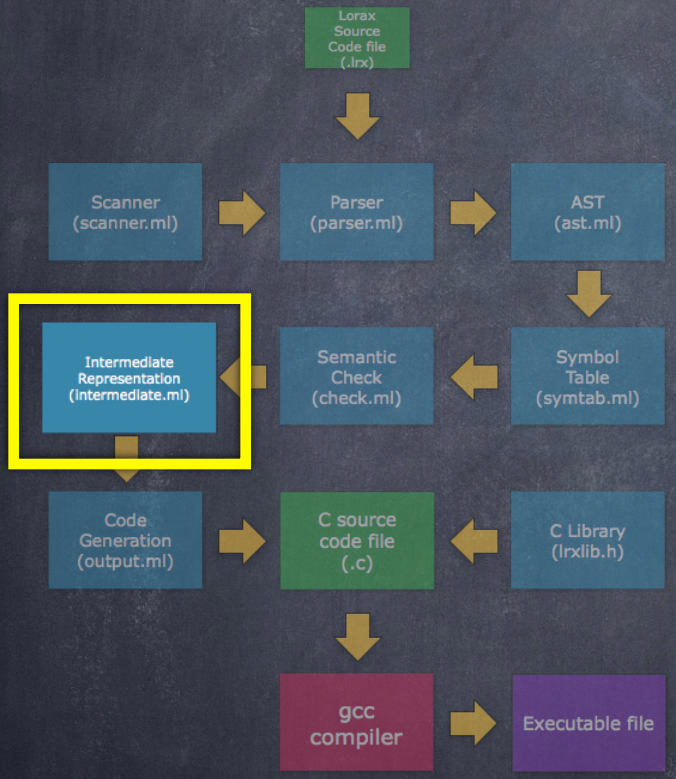
How it Works: Intermediate Representation



Unique to Lorax

- Unraveling the tree data structure from the checked AST to a series of instructions was very difficult.
- Understanding when our target language would require pointers, double pointers, triple pointers, dereference, or require address operators was very challenging.
- In Lorax this module in conjunction with output.ml and lrxlib proved the most challenging.

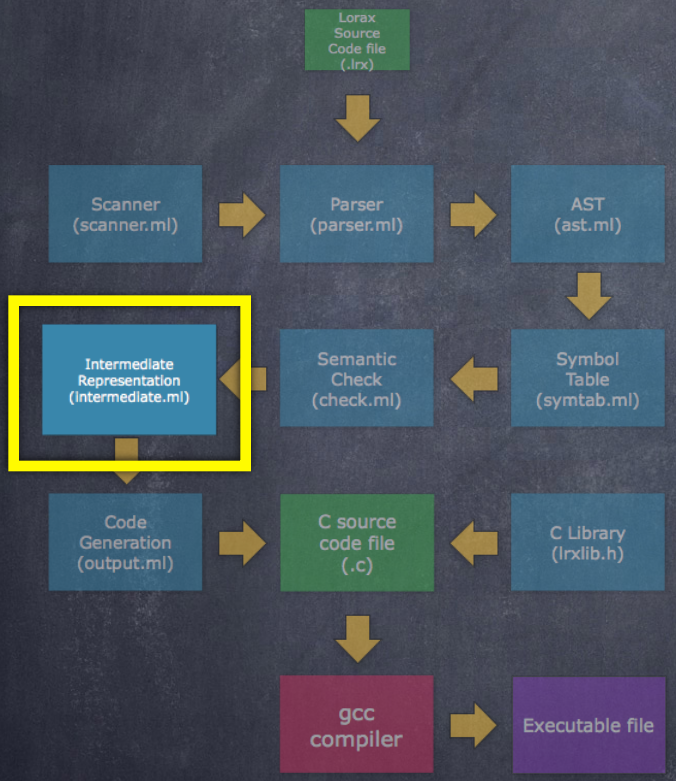
How it Works: Intermediate Representation



Code snippet critical to tree unraveling and creation.

```
let gen_tmp_internal child tree_type child_number child_array =
  [Ir_Internal(child_array, child_number, child)]

let rec gen_tmp_internals children tree_type array_access child_array =
 match children with
    [] -> []
  | head :: tail -> gen_tmp_internal head tree_type array_access child_array @ gen_tmp_internals
tail tree_type (array_access + 1) child_array

let gen_tmp_child child tree_type tree_degree =
  if (is_atom child) then
    let tmp_root_data = (gen_tmp_var tree_type 0) in
    let d =
    (match tree_type with
        Lrx_Atom(a) -> a
      | Lrx_Tree(t) -> raise(Failure "Tree type as tree data item. (Error 3)")) in
    let tmp_leaf_children = (gen_tmp_var (Lrx_Tree({datatype = d; degree =
Int_Literal(tree_degree)}))) 0) in
    let tmp_leaf_root = (gen_tmp_var (Lrx_Tree({datatype = d; degree =
Int_Literal(tree_degree)}))) 0) in
    ([Ir_At_Ptr(tmp_root_data);
      Ir_Ptr(tmp_root_data, child);
      Ir_Leaf(tmp_leaf_children, tree_degree);
      Ir_Decl(tmp_leaf_root);
      Ir_Tree_Destroy(tmp_leaf_root);
      Ir_Expr(Ir_Tree_Literal(tmp_leaf_root, tmp_root_data, tmp_leaf_children))], tmp_leaf_root)
  else
    ([], child)

let rec gen_tmp_children children tree_type tree_degree =
  match children with
  [] -> []
 | head :: tail -> gen_tmp_child head tree_type tree_degree :: gen_tmp_children tail tree_type
tree_degree

let gen_tmp_tree tree_type tree_degree root children_list tmp_tree =
  let children = gen_tmp_children children_list tree_type tree_degree in
  let (decls, tmp_children) = (List.fold_left (fun (a, b) (c, d) -> ((c @ a), (d :: b))) ([],[])
(List.rev children)) in
  let d =
  (match tree_type with
      Lrx_Atom(a) -> a
    | Lrx_Tree(t) -> raise(Failure "Tree type as tree data item. (Error 1)")) in
  let child_array = gen_tmp_var (Lrx_Tree({datatype = d; degree = Int_Literal(tree_degree)})) 0
in
  let internals = gen_tmp_internals tmp_children tree_type 0 child_array in
  let tmp_root_ptr = gen_tmp_var tree_type 0 in
  decls @ [Ir_Child_Array(child_array, tree_degree)] @ internals @ [Ir_At_Ptr(tmp_root_ptr);
Ir_Ptr(tmp_root_ptr, root)] @ [Ir_Expr(Ir_Tree_Literal(tmp_tree, tmp_root_ptr, child_array))]
```

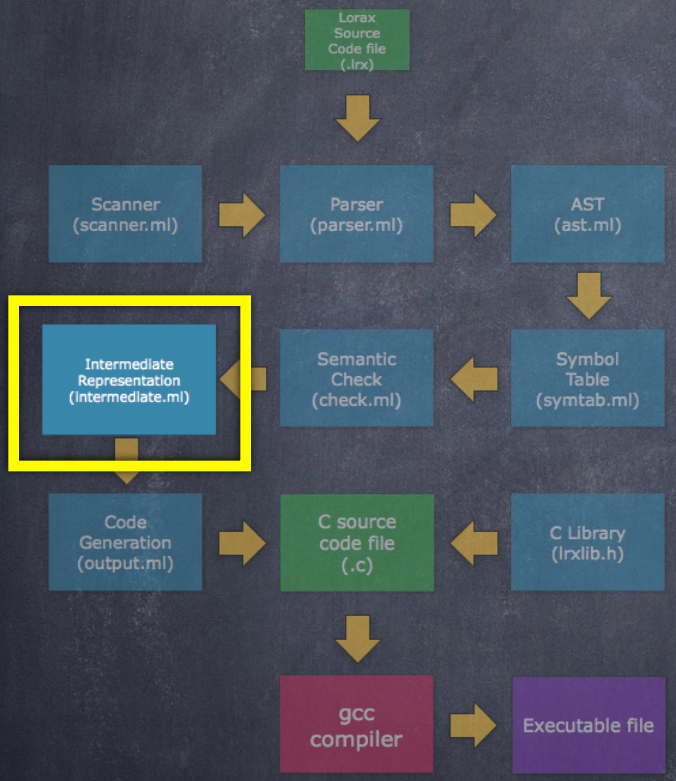
How it Works: Intermediate Representation



Intermediate Representation (part 1 of 4)

```
inter_pgrm: Intermediate.ir_program =
  {Intermediate.ir_globals = [];
   ir_headers =
    [{Intermediate.ir_name = "main"; ir_ret_type = Ast.Lrx_Atom Ast.Lrx_Int;
      ir_formals = []}];
   ir_bodies =
    [{Intermediate.ir_header = (Ast.Lrx_Atom Ast.Lrx_Int, "main", []);
      ir_vdecls =
       [Intermediate.Ir_Decl
         ("t",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          1, 0);
        Intermediate.Ir_Decl
         ("__tmp_tree_datatype_int_degree_2",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          5, 0);
        Intermediate.Ir_Decl ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 3, 0);
        Intermediate.Ir_Decl ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 4, 0);
        Intermediate.Ir_Decl ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 2, 0);
        Intermediate.Ir_At_Ptr ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 9, 0);
        Intermediate.Ir_Decl
         ("__tmp_tree_datatype_int_degree_2",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          11, 0);
        Intermediate.Ir_At_Ptr ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 6, 0);
        Intermediate.Ir_Decl

...
```

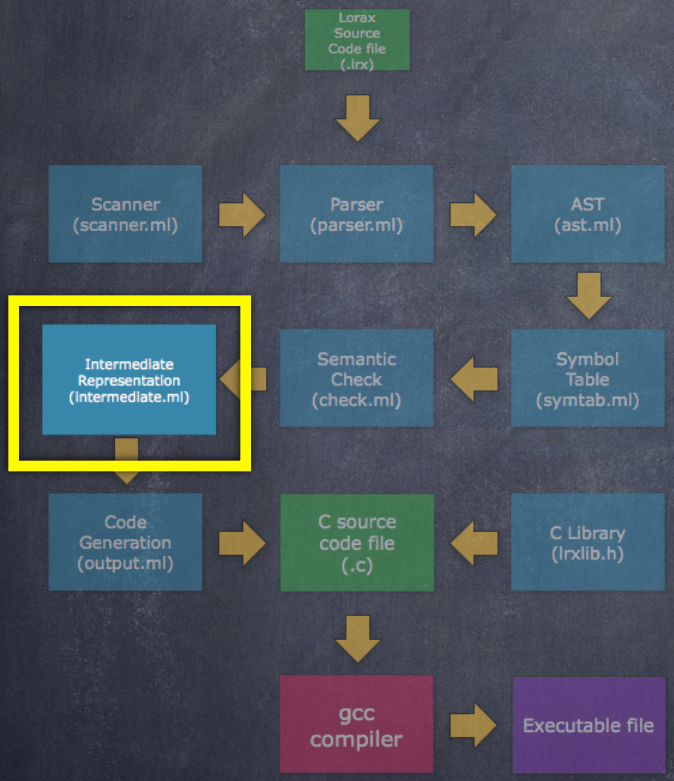
How it Works: Intermediate Representation



Intermediate Representation (part 2 of 4)

```
...
        ("__tmp_tree_datatype_int_degree_2",
         Ast.Lrx_Tree
          {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
         8, 0);
       Intermediate.Ir_At_Ptr ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 13, 0);
       Intermediate.Ir_Decl ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 1, 0);
       Intermediate.Ir_Decl ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 0, 0)];
     ir_stmts =
      [Intermediate.Ir_Expr
       (Intermediate.Ir_Int_Literal
        (("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 3, 0), 2));
       Intermediate.Ir_Expr
        (Intermediate.Ir_Int_Literal
         (("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 4, 0), 3));
       Intermediate.Ir_Expr
        (Intermediate.Ir_Int_Literal
         (("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 2, 0), 1));
       Intermediate.Ir_Ptr (("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 9, 0),
        ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 3, 0));
       Intermediate.Ir_Leaf
        (("__tmp_tree_datatype_int_degree_2",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          10, 0),
         2);
...
```

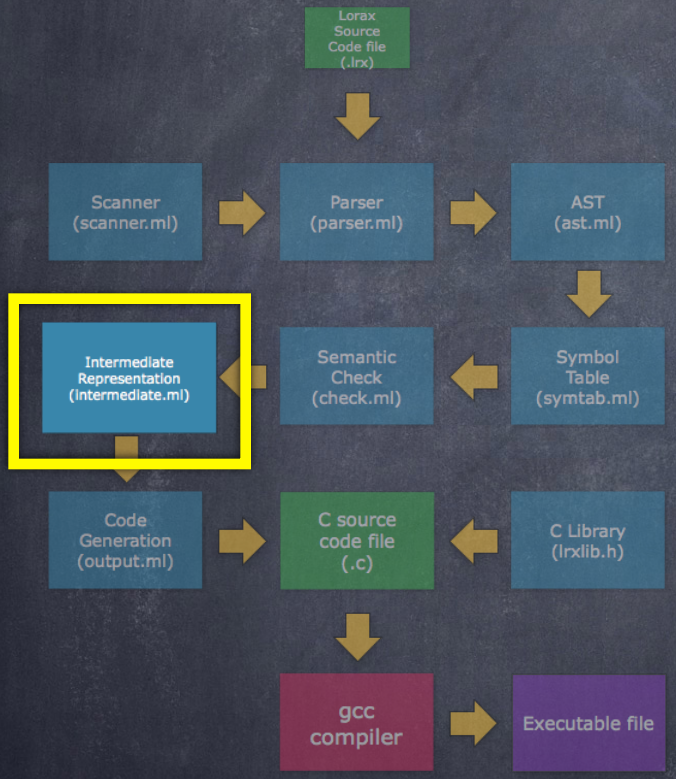
How it Works: Intermediate Representation



Intermediate Representation (part 3 of 4)

```
...

      Intermediate.Ir_Expr
       (Intermediate.Ir_Tree_Literal
         (("__tmp_tree_datatype_int_degree_2",
           Ast.Lrx_Tree
            {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
           11, 0),
         ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 9, 0),
         ("__tmp_tree_datatype_int_degree_2",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          10, 0)));
      Intermediate.Ir_Ptr (("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 6, 0),
       ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 4, 0));
      Intermediate.Ir_Leaf
       (("__tmp_tree_datatype_int_degree_2",
         Ast.Lrx_Tree
          {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
         7, 0),
       2);
      Intermediate.Ir_Expr
       (Intermediate.Ir_Tree_Literal
         (("__tmp_tree_datatype_int_degree_2",
           Ast.Lrx_Tree
            {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
           8, 0),
         ("__tmp_int", Ast.Lrx_Atom Ast.Lrx_Int, 6, 0),
         ("__tmp_tree_datatype_int_degree_2",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          7, 0)));
...
```

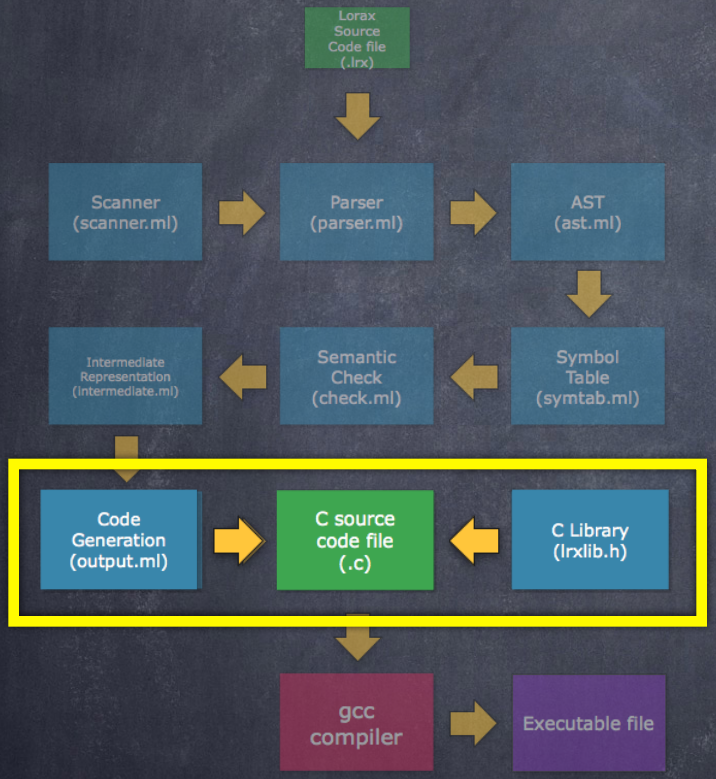
How it Works: Intermediate Representation



Intermediate Representation (part 4 of 4)

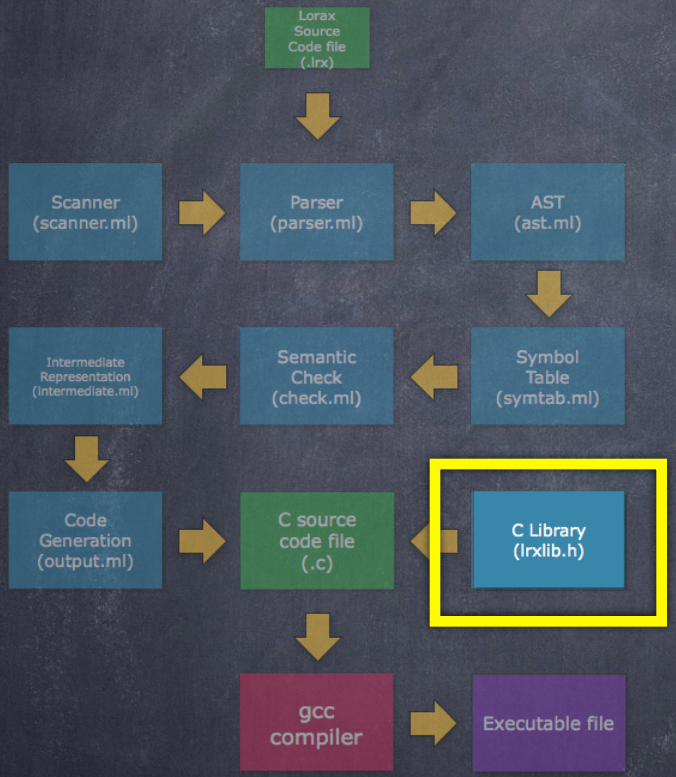
```
...

        Intermediate.Ir_Child_Array
         (("__tmp_tree_datatype_int_degree_2",
           Ast.Lrx_Tree
            {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
           12, 0),
         2);
        Intermediate.Ir_Internal
         (("__tmp_tree_datatype_int_degree_2",
           Ast.Lrx_Tree
            {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
           12, 0),
         0,
         ("__tmp_tree_datatype_int_degree_2",
          Ast.Lrx_Tree
           {Ast.datatype = Ast.Lrx_Int; degree = Ast.Int_Literal 2},
          11, ...));
        ...];
      ir_destroys = ...};
    ...]}
```

How it Works: Code Generation



- Code Generation is responsible for writing the intermediate representation list to valid c syntax.
- We added comments in our generated c code to assist with debugging.

How it Works: lrxlib.h



Lorax Tree C Code Data Structure

```
tree * t_1 = lrx_declare_tree(_INT_, 2);
```

```
typedef struct tree{
    int degree;
    Atom datatype; /* enum: bool, int, float, char */
    Root root; /* union: char, int, bool, float */
    struct tree **children;
    struct tree *parent;
    bool leaf; /* leaf == childless */
    int *count; /* reference count (smart pointer) */
} tree;
```

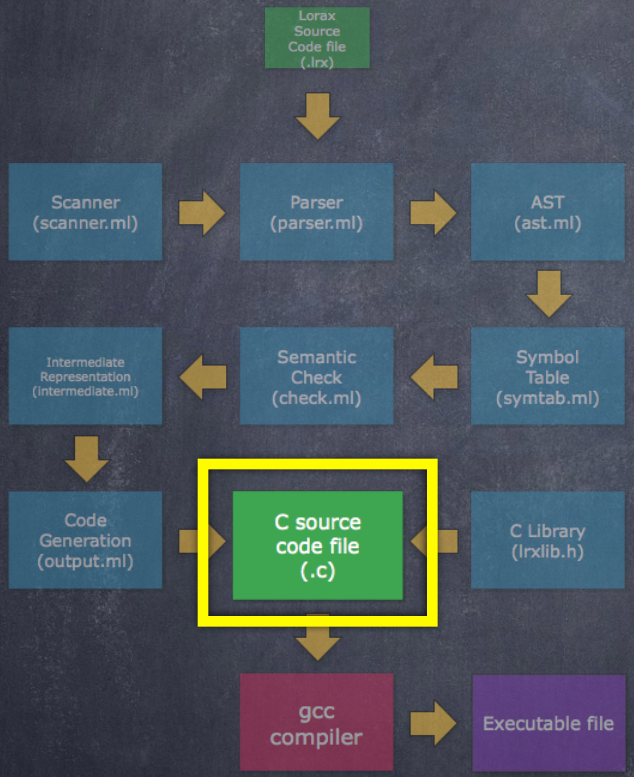
```
tree *
__tmp_tree_datatype_int_degree_2_11 =
lrx_declare_tree(_INT_, 2); /* Ir_Decl
*/
```

```
tree *
__tmp_tree_datatype_int_degree_2_8 =
lrx_declare_tree(_INT_, 2); /* Ir_Decl
*/
```

```
struct tree
```

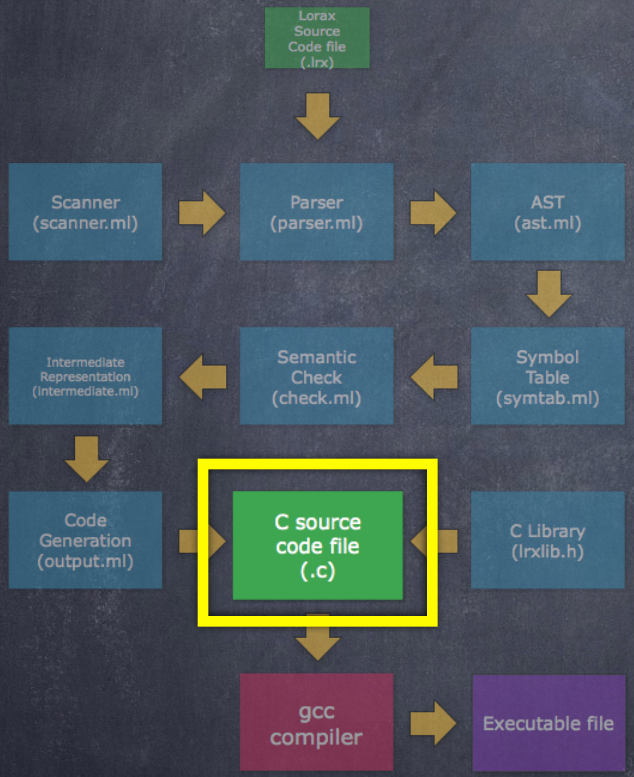
```
struct tree
```

How it Works: Compiler Output (1 of 3)



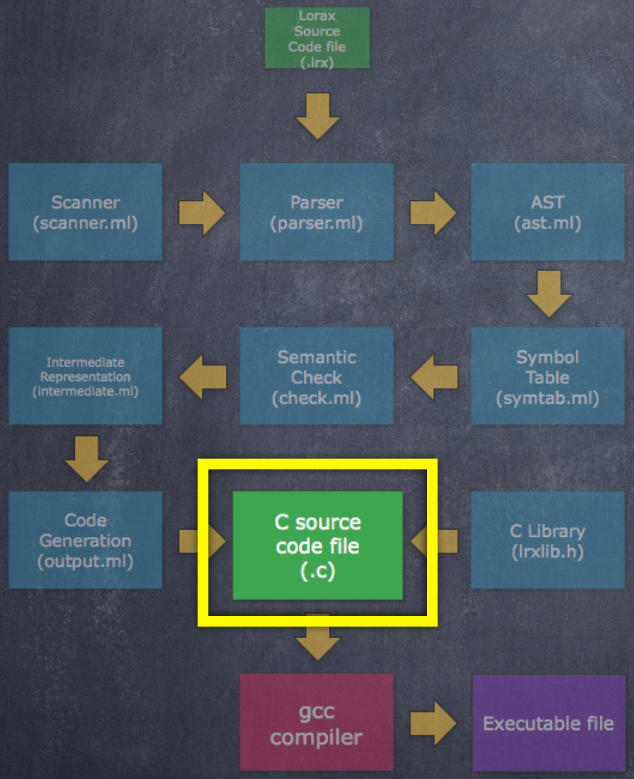

```c
#include "lrxlib.h"
int main();

int main()
{
tree * t_1 = lrx_declare_tree(_INT_, 2); /* Ir_Decl */
tree * __tmp_tree_datatype_int_degree_2_5 = lrx_declare_tree(_INT_,
2); /* Ir_Decl */
int __tmp_int_3 = 0; /* Ir_Decl */
int __tmp_int_4 = 0; /* Ir_Decl */
int __tmp_int_2 = 0; /* Ir_Decl */
int *__tmp_int_9 = NULL; /* Ir_At_Ptr */
tree * __tmp_tree_datatype_int_degree_2_11 = lrx_declare_tree(_INT_,
2); /* Ir_Decl */
int *__tmp_int_6 = NULL; /* Ir_At_Ptr */
tree * __tmp_tree_datatype_int_degree_2_8 = lrx_declare_tree(_INT_,
2); /* Ir_Decl */
int *__tmp_int_13 = NULL; /* Ir_At_Ptr */
int __tmp_int_1 = 0; /* Ir_Decl */
int __tmp_int_0 = 0; /* Ir_Decl */

__tmp_int_3 = 2;

__tmp_int_4 = 3;

__tmp_int_2 = 1;

__tmp_int_9 = &__tmp_int_3; /* Ir_Ptr */
tree * __tmp_tree_datatype_int_degree_2_10[2]; /* Ir_Leaf */
__tmp_tree_datatype_int_degree_2_10[1] = NULL; /* c_of_leaf */
__tmp_tree_datatype_int_degree_2_10[0] = NULL; /* c_of_leaf */

...
```

How it Works: Compiler Output (2 of 3)



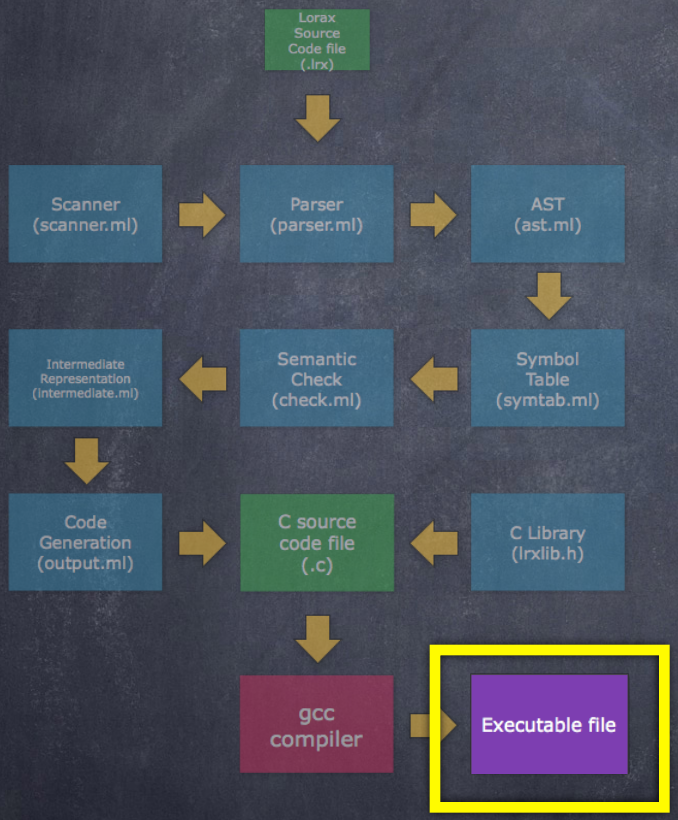

```
...

lrx_define_tree(__tmp_tree_datatype_int_degree_2_11,
__tmp_int_9, __tmp_tree_datatype_int_degree_2_10);

__tmp_int_6 = &__tmp_int_4; /* Ir_Ptr */
tree * __tmp_tree_datatype_int_degree_2_7[2]; /* Ir_Leaf */
__tmp_tree_datatype_int_degree_2_7[1] = NULL; /* c_of_leaf */
__tmp_tree_datatype_int_degree_2_7[0] = NULL; /* c_of_leaf */

lrx_define_tree(__tmp_tree_datatype_int_degree_2_8,
__tmp_int_6, __tmp_tree_datatype_int_degree_2_7);

tree * __tmp_tree_datatype_int_degree_2_12[2]; /*
Ir_Child_Array */
/* Filling with NULL preemptively */
__tmp_tree_datatype_int_degree_2_12[1] = NULL; /* c_of_leaf */
__tmp_tree_datatype_int_degree_2_12[0] = NULL; /* c_of_leaf */

__tmp_tree_datatype_int_degree_2_12[0] =
__tmp_tree_datatype_int_degree_2_11; /* Ir_Internal */
__tmp_tree_datatype_int_degree_2_12[1] =
__tmp_tree_datatype_int_degree_2_8; /* Ir_Internal */
__tmp_int_13 = &__tmp_int_2; /* Ir_Ptr */
lrx_define_tree(__tmp_tree_datatype_int_degree_2_5,
__tmp_int_13, __tmp_tree_datatype_int_degree_2_12);

lrx_assign_tree_direct(&t_1,
&__tmp_tree_datatype_int_degree_2_5);

...
```

How it Works: Compiler Output (3 of 3)




```
...

lrx_print_tree(t_1);

goto __LABEL_0;
__LABEL_1:
return __tmp_int_0;
__LABEL_0:
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_8);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_11);
lrx_destroy_tree(__tmp_tree_datatype_int_degree_2_5);
lrx_destroy_tree(t_1);

goto __LABEL_1;
}
```

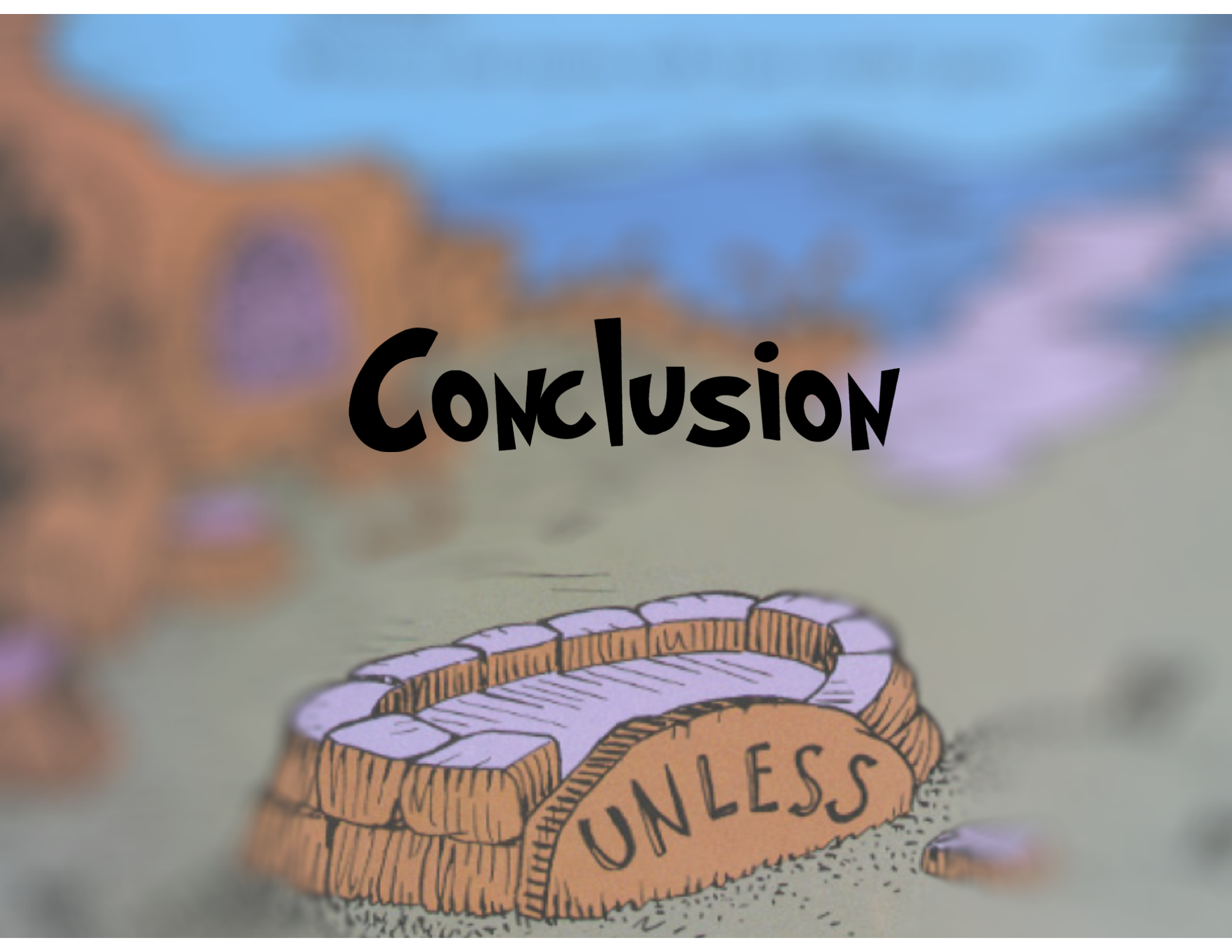
How it Works: Execution / Memory Management



Valgrind Memory Test

```
vagrant@precise64:/vagrant/PLT$ valgrind ./a.out
==1150== Memcheck, a memory error detector
==1150== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==1150== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==1150== Command: ./a.out
==1150==

1[2[null,null],3[null,null]]

==1150==
==1150== HEAP SUMMARY:
==1150==     in use at exit: 0 bytes in 0 blocks
==1150==   total heap usage: 12 allocs, 12 frees, 272 bytes allocated
==1150==
==1150== All heap blocks were freed -- no leaks are possible
==1150==
==1150== For counts of detected and suppressed errors, rerun with: -v
==1150== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Conclusion



Conclusion

We were able to accomplish most of the major goals we designed for our language!

There are some features that are still in development and can be found partially completed in the code base. They are:

- Pop operator
- Continue and Break
- Short circuit evaluation

Conclusion

Lessons Learned

- Do not wait. Start early, even if you don't feel like you know what you are doing. You will learn while doing.
- Compiler writing is fun!
- OCaml is the greatest. Its type checking is often smarter than you are.
- Understand your AST to target language thoroughly. Using different varieties of pointer referencing provided a significant technical challenge that may have been avoided with more planning.
- Steal from the best. MicroC and Dara Hazeghi's (2011) strlang was our template and greatest source of instruction.
- Use the debugger.

LORAX

The Language that Speaks for the Trees

http://bit.ly/theloraxcode

http://bit.ly/theloraxmanual