SEE++

Final Project Report

By Adar Tulloch, Vishrut Tiwari, Winston Zhang, and Jack LaVelle

Contents

1. Introduction and motivation:

SEE++ is a language capable of making digital art. Starting with pixels as the base building block users will be able to expand these into lines, shapes, 2D custom graphics, and more. They can combine with other users' artwork and build on previous projects. This project is inspired by the programming language Processing and we will follow the object oriented programming paradigm mixed with the simplified C like syntax.

2. Language Reference Manual

Lexical conventions

Tokens: the types of See++ tokens are described below: keywords, comments, and identifiers.

Keywords: the following are reserved keywords

1. int
2. char
3. float
4. bool
5. if
6. else
7. else if
8. for

9. main
10. Void
11. Canvas
12. Pixel
13. Point
14. ShoeHorn()
15. Circle

Comments: Single line comments are introduced by //, multi-line comments are introduced by /* and terminated by */.

Identifiers: Identifiers are a case-sensitive sequence of letters, numbers, and the underscore symbol. The first character of an identifier must be a letter or underscore. Characters must be members of the Unicode character set.

Data Types:

See++ supports fundamental types such as the Integer, String, and Boolean, as well as advanced, language-specific types called “Pixel” and “Canvas” that work to render art to the screen.

Integers (`int`) 32 bit and signed data

Characters(`char`) 8 bit letters

Strings (`String`) sequence of letters stored in the heap

Boolean (declared and called `boolean`) singular bit of data used to assign true or false value

Pixels (`Pixel(int)`) 8 bits of data to represent grayscale

Canvas(`Canvas(float,float)`) made up of two integers of 8 bits to initialize screen size

Operators:

Operations	Storage
`!`	Logical NOT operator
`*  /`	Multiplication and division
`+  -`	Addition and subtraction
`>  <  >=  <=`	Greater than, less than, greater than or equal to, less than or equal to
`==  !=`	Equality, inequality
`==  !=`	Logical AND, logical OR
`=`	Assignment operation
`-> append()`	Adds pixel to canvas
`-> append().circle`	Adds circle to canvas

Note: the above operations are listed in precedence with the operators at the top having the most precedence.

Functionality and Syntax

Function calls: See++ follows C-like function calls:

```
type name(args){
...
}
```

The `type` here is the return value and the `name` is the function name. In SEE++ the first function sets up the canvas and is called set up and does not return anything back:

```
void setup(args){
canvas(500,500)
}
```

We also have another function called `draw()` which is the main function that continually runs code contained within its parentheses.

List of built-in functions:

Functions	Description
`point(x,y,pixel)`	Places a point on the canvas where `x` and `y` are the coordinates and pixel carries the grayscale value
`Circle(x,y,r)`	The circle takes in three floats
`draw(Canvas(float,float),"file.svg")`	Draw has a file name inputted into it which has to be an .svg file we can write to. `Canvas` as seen before is an object with dimension of how big the .svg file is.
`printf(float)`	Prints out a float
`print(int)`	Prints out a string value given an int
`printbig(int)`	Takes in an int and returns ascii

Variable declarations: similar to C ex: `int a = 5;`

Scope: Variables can be declared globally if they are located outside of a function declaration. Our function declarations still work the same as in MicroC where they are defined towards the top. However variables declared inside of the draw and methods otherwise known as lexically scoped will not be known outside that scope.

Statements

Statements are sequences of code that are executed in order unless otherwise specified.

Blocks: statements may be grouped in blocks, delineated by braces `{}`.

Conditionals: conditional statements follow similarly from C and include **if, else,** and **else if**. Conditional statements must be followed by a block of code that is executed when the condition is met.

```
int main()
{
  if (true) print(42); else print(8);
  print(17);
  return 0;
```

```
}
```

To resolve “dangling-else” problem, See++ matches **else** with the closest elseless **if.**

For Loop: for loops are a form of iteration and syntax follows similarly from Java. A for loop contains initialization, a condition, and an updating clause, and must be followed by a code block.

```
int main()
{
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(i);
  }
  print(42);
  return 0;
}
```

While: while loops are another form of iteration similar to for loops. While loops contain a condition and a block of code to be executed while the condition is still met. The following is equivalent to the for loop example above. int foo(int a)

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

```
  }
  return j;
}

int main()
{
  print(foo(7));
  return 0;
}
```

Return Statement: the return statement returns a particular value from a function and exits the function. All non-void functions must return a value that matches the function signature. Void functions may use the simple return keyword to indicate the termination of a function.

Example:

```
int main(){
    Canvas can = Canvas(100.0,100.0);
     Point pt = Point(500.0, 500.0);
     Pixel p = Pixel(pt);
     can -> append() c;
     drawcircle(can, "CANCIRshape.svg");
     return 0;
}
```

[Need to add circleCanvas example]

3. Project Plan:

Manager (the boss): Adar

Language Guru - Winston

System Architect - Jack

Tester: Vishrut

The planning stage:

We mostly met on Tuesday during Hao's officer hours every week. He was super helpful in guiding us along the way! We got a good portion of work done during this time and then zoomed on various days throughout the week to update our progress and add new features. We started with just understanding how codegen would output our LLVM IR. We felt as though if we understood codegen well, the rest would fall into place.

Software Development Environment

Our technology stack includes OCaml and its lexer and parser tools (ocamllex and ocamlyacc), ocamlbuild, opam, LLVM, Docker, and git. We used the *columbiasedwards/plt* Docker image to load a container with all our necessary dependencies. This virtual environment proved to be extremely useful to build and test throughout all of our team member's machines. For the code review process, we checked in pull requests to Github as needed, and collectively reviewed and made changes as needed.

The development stage:

We added new features and walked through each step linearly. Our whole group wanted an understanding of how each part worked so most of the time

we broke off into pairs and implemented a new feature from the scanner all the way to the external C files. The scanner and parser were implemented in roughly a week, and the rest of our time was spent on codegen and linking the external C libraries.

The testing stage:

Writing test files mostly to see if our feature worked were created to test that feature's functionality so everyone played a part in th at as well. So this was done alongside the development stage and we wrote a few more tests once we were finally done to see if we missed a few edge cases.

Our testing suite was built on top of Microc's testing framework. For our language-specific features, which include drawing and rendering SVG files, we manually tested by running our code and visualizing the SVG output in a browser to ensure that our compiler worked as expected.

Timeline:

October 11th: Created repo Github

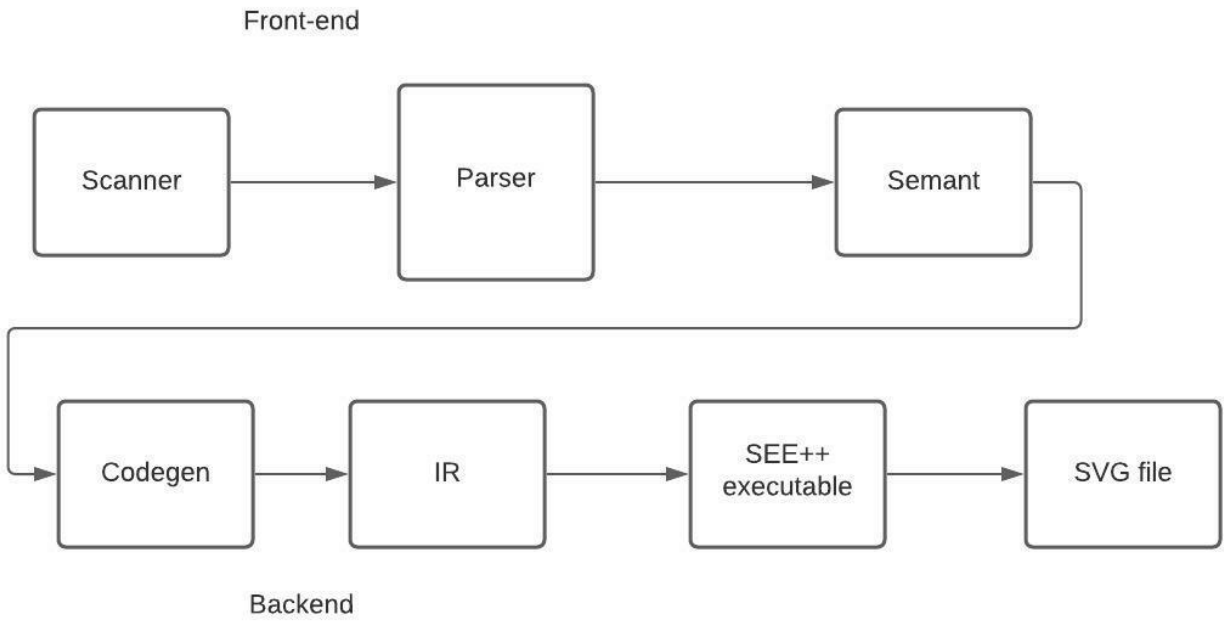
November 5th: Completed Hello world

November 15th: Implemented Canvas SVG file with a point

December 1st: Added pixels

December 5th: First Hello World program using pixels

December 15th: Added a circle



Who did what?

As mentioned before we all walked through each part together and troubleshot different parts whenever one of us got stuck. It would be hard to purely associate one part with one person as we all were greedy and wanted to understand what each part did. Pair programming here was crucial as pairs of us would walk through implementing a pixel and a circle. Once we got one to work we could easily get the .ll file code and compare with the broken version and see what we needed to fix. For our project in specific programming like this was key and worked very well. - Adar, Winston, Jack, Vishrut

Lessons learned:

Adar: Test! Always test early and often! My favorite Edwards quote from the semester is “untested code never works, Schrodinger’s cat is always dead.” Make sure you find a good group, also. I’m thankful for my other three teammates for their ideas just when my brain seemed to stop working. Also, rely heavily on the TA’s, they are often really helpful with roadmapping.

Vishrut: I gained a deeper appreciation of working with teammates and planning a project out very meticulously. It was important to sit down with your teammates or the TA and talk about your vision for your project. I also learned that the compiler is fussy and you have to be very careful about what you change. Most things we take for granted while learning how to program are actually very difficult to make on a low level and compiler design is really a dragon that a knight has to slay.

Jack: There were multiple times during this project that was very easily made into a learning moment. The first was one that I did not know occurred until long after which was to start early and chip away slowly at the project. This can be difficult as there are always assignments from other classes that are due much sooner so it is important to meet often with the team and with the TA. This brings me to my second lesson learned and that was how to function in a team and especially in cases where you need to work off of the understanding of your partners to achieve your own bit. This project was really a project of projects where each stage of the compiler was its own beast.

Winston: The project is like a Greek tragedy: we were told what would go wrong, we tried everything to prevent it from happening, and despite our best

efforts, it still happens. We started the project, full of hope and ambition. As we worked on the project, reality settled in, and we soon realized that what we wanted initially was simply not feasible. My main takeaway was to be realistic with our project ideas, and take advice from our professor and TAs. Additionally, there are often times when we try to work on something without fully understanding the functionality of what we're using. This caused us a lot of headaches and took extensive time to bug fix. This project was not only a lesson on compilers but also a lesson on teamwork.

5. Appendix

ast.ml

```
(* See++ Abstract Syntax Tree and functions for printing it
    Authors: Adar Tulloch, Vishrut Tiwari, Winston Zhang, and Jack LaVelle *)
type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater | Geq
|
          And | Or | Shoehorn | ShoehornCircle

type uop = Neg | Not

type typ = Int | Bool | Float | Void | Char | String | Point | Pixel | Canvas |
Circle | CanvasCircle

type bind = typ * string

type expr =
    Literal of int
  | Fliteral of string
  | BoolLit of bool
  | CharLit of char
  | StringLit of string
  | Id of string
  | Binop of expr * op * expr
  | Field of string * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

type stmt =
    VDecl of typ * string
  | VDeclAssign of typ * string * expr
  | Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
```

```
type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    body : stmt list;
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"
  | Shoehorn -> "-> append()"
  | ShoehornCircle -> "-> append().circle"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"

let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | Char -> "char"
```

```
  | String -> "String"
  | Point -> "Point"
  | Pixel -> "Pixel"
  | Canvas -> "Canvas"
  | Circle -> "Circle"
  | CanvasCircle -> "CanvasCircle"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Fliteral(l) -> l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | CharLit(l) -> String.make 1 l
  | StringLit(l) -> l
  | Field(s,f) -> s ^ "." ^ string_of_expr f
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

  let rec string_of_stmt = function
    VDecl(t, i) -> string_of_typ t ^ " " ^ i ^ "\n"
  | VDeclAssign(t, i, e) -> string_of_typ t ^ " " ^ i ^ " = " ^ string_of_expr e
^ "\n"
  | Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
```

```
let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  (* String.concat "" (List.map string_of_vdecl fdecl.locals) ^ *)
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

Codegen.ml

```
(* SEE++ Code generation: translate takes a semantically checked AST and
    produces LLVM IR

    Authors: Adar Tulloch, Vishrut Tiwari, Winston Zhang, and Jack LaVelle

Detailed documentation on the OCaml LLVM library:

http://llvm.org/docs/tutorial/index.html
http://llvm.moe/
http://llvm.moe/ocaml/


*)
module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)
```

```
(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "SEEPP" in

  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context
  and str_t    = L.pointer_type (L.i8_type context)
  and void_t     = L.void_type   context in
  let float_t    = L.double_type context in
  let ptstruct_t = L.struct_type context [| float_t ; float_t |] in
  let pstruct_t  = L.struct_type context [| ptstruct_t |] in
  let cstruct_t  = L.struct_type context [| ptstruct_t; float_t |] in

  let canvasnode_t = L.named_struct_type context "canvasnode" in
  ignore(L.struct_set_body canvasnode_t [| L.pointer_type (canvasnode_t) ;
      (L.pointer_type pstruct_t) |] false);

  let canvascirclenode_t = L.named_struct_type context "canvascirclenode" in
  ignore(L.struct_set_body canvascirclenode_t [| L.pointer_type
(canvascirclenode_t) ;
      (L.pointer_type cstruct_t) |] false);

  let canvas_t = L.struct_type context [| float_t ; float_t ;
      (L.pointer_type canvasnode_t) |]
  in

  let canvascircle_t = L.struct_type context [| float_t ; float_t ;
      (L.pointer_type canvascirclenode_t) |]
  in

  (* Return the LLVM type for a See++ type *)
  let ltype_of_typ = function
      A.Int   -> i32_t
```

```
  | A.Bool  -> i1_t
  | A.Float -> float_t
  | A.Void  -> void_t
  | A.String -> str_t
  | A.Char  -> i8_t
  | A.Point -> ptstruct_t
  | A.Pixel -> pstruct_t
  | A.Circle -> cstruct_t
  | A.Canvas -> canvas_t
  | A.CanvasCircle -> canvascircle_t

in

(* Create a map of global variables after creating each *)
let global_vars : L.llvalue StringMap.t =
  let global_var m (t, n) =
    let init = match t with
        A.Float -> L.const_float (ltype_of_typ t) 0.0
      | _ -> L.const_int (ltype_of_typ t) 0
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals in


let printf_t : L.lltype =
    L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func : L.llvalue =
    L.declare_function "printf" printf_t the_module in
let printbig_t : L.lltype =
    L.function_type i32_t [| i32_t |] in
let printbig_func : L.llvalue =
    L.declare_function "printbig" printbig_t the_module in
let draw_t : L.lltype =
    L.function_type i32_t [| canvas_t ; str_t |] in
let draw_func : L.llvalue =
    L.declare_function "draw" draw_t the_module in

let drawcircle_t : L.lltype =
    L.function_type i32_t [| canvascircle_t ; str_t |] in
let drawcircle_func : L.llvalue =
```

```
      L.declare_function "drawcircle" drawcircle_t the_module in

  let ptcons_t : L.lltype =
      L.function_type ptstruct_t [|float_t; float_t|] in
  let ptcons_func : L.llvalue =
      L.declare_function "Point" ptcons_t the_module in
  let ccons_t : L.lltype =
      L.function_type pstruct_t [|ptstruct_t|] in
  let ccons_func : L.llvalue =
      L.declare_function "Pixel" ccons_t the_module in
  let canvascons_t : L.lltype =
      L.function_type canvas_t [|float_t; float_t; (* L.pointer_type canvasnode_t
*)|] in
  let canvascons_func : L.llvalue  =
      L.declare_function "Canvas" canvascons_t the_module in

  let circlecons_t : L.lltype =
      L.function_type cstruct_t [|ptstruct_t; float_t; (* L.pointer_type cstruc
*)|] in
  let circlecons_func : L.llvalue =
      L.declare_function "Circle" circlecons_t the_module in
  let canvascirclecons_t : L.lltype =
      L.function_type canvascircle_t [|float_t; float_t; (* L.pointer_type
canvascirclenode_t *)|] in
  let canvascirclecons_func : L.llvalue  =
      L.declare_function "CanvasCircle" canvascirclecons_t the_module in

  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
    let function_decl m fdecl =
      let name = fdecl.sfname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
      in
      let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
      StringMap.add name (L.define_function name ftype the_module, fdecl) m in
    List.fold_left function_decl StringMap.empty functions in
```

```
(* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.sfname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
  and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
  and str_format_str = L.build_global_stringptr "%s\n" "fmt" builder in

  (* Construct the function's "locals": formal arguments and locally
     declared variables.  Allocate each on the stack, initialize their
     value, if appropriate, and remember their values in the "locals" map *)
  let local_vars =
    let add_formal m (t, n) p =
      L.set_value_name n p;
      let local = L.build_alloca (ltype_of_typ t) n builder in
      ignore (L.build_store p local builder);
       StringMap.add n local m

    (* Allocate space for any locally declared variables and add the
     * resulting registers to our map *)
    in

  List.fold_left2 add_formal StringMap.empty fdecl.sformals
    (Array.to_list (L.params the_function)) in

  (* Return the value for a variable or formal argument.
     Check local names first, then global names *)
  let lookup n locals = try StringMap.find n locals
                with Not_found -> StringMap.find n global_vars
  in


  let mem_to_ind ty = match ty  with
    _ -> List.fold_left (fun m (name, ind) -> StringMap.add name ind m)
               StringMap.empty [("ep1",0); ("ep2",1); ("cp1",2);
                   ("cp2",3); ("x",0); ("y",1)]
  in
```

```
(* Construct code for an expression; return its value *)
let rec expr builder locals ((_, e) : sexpr) = match e with
    SLiteral i   -> L.const_int i32_t i
  | SBoolLit b   -> L.const_int i1_t (if b then 1 else 0)
  | SFliteral l  -> L.const_float_of_string float_t l
  | SCharLit l   -> L.const_int i8_t (Char.code l)
  | SStringLit l -> L.build_global_stringptr l "str" builder
  | SNoexpr      -> L.const_int i32_t 0
  | SId s        -> L.build_load (lookup s locals) s builder
  | SAssign (s, e) -> let e' = expr builder locals e in
                      ignore(L.build_store e' (lookup s locals) builder); e'
  | SField(id,sx) ->
     let getI t n =
       try StringMap.find n (mem_to_ind t)
       with Not_found -> raise(Failure("member not found"))in
     let getNextVal o t n = L.build_struct_gep o (getI t n) n builder in
     let rec eval out t = function
         SField(sid, sf)-> eval (getNextVal out t sid)
             (L.type_of(getNextVal out t sid)) (snd sf)
       | SId sid ->
             let ref = L.build_struct_gep out (getI t sid) sid builder in
             L.build_load ref sid builder
       | SAssign(s,e) ->
             let ref = L.build_struct_gep out (getI t s) s builder in
             let e' =  expr builder locals e in
             ignore(L.build_store e' ref builder); e'
       | _ -> raise(Failure("invalid field usage"))
     in eval (lookup id locals) (L.type_of (lookup id locals)) (snd sx)
  | SBinop ((A.Float,_ ) as e1, op, e2) ->
     let e1' = expr builder locals e1
     and e2' = expr builder locals e2 in
     (match op with
         A.Add     -> L.build_fadd
       | A.Sub     -> L.build_fsub
       | A.Mult    -> L.build_fmul
       | A.Div     -> L.build_fdiv
       | A.Mod     -> L.build_srem
       | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
       | A.Neq     -> L.build_fcmp L.Fcmp.One
```

```
            | A.Less    -> L.build_fcmp L.Fcmp.Olt
            | A.Leq     -> L.build_fcmp L.Fcmp.Ole
            | A.Greater -> L.build_fcmp L.Fcmp.Ogt
            | A.Geq     -> L.build_fcmp L.Fcmp.Oge
            | _ -> raise (Failure ("illegal usage of operator " ^
                                   (A.string_of_op op) ^ " on float"))
          ) e1' e2' "tmp" builder
      | SBinop((A.Canvas, _) as can, op, crv) ->
          let (_,can_s) = (match (snd can) with
              SId s -> (expr builder locals can, s)
              |_-> raise(Failure "improper usage of shoehorn - canvas"))
          and (_,px_s) = (match (snd crv) with
              SId s -> (expr builder locals crv,s)
              |_->raise(Failure "improper usage of shoehorn - pixel")) in
          (match op with
              A.Shoehorn   ->
                (* construct new node, add it to front of list *)
                let newnode = L.build_alloca canvasnode_t "newnode" builder in
                let next_node_ptr = L.build_struct_gep newnode 0 "new_pixel"
builder in
                ignore(L.build_store (L.const_null (L.pointer_type canvasnode_t))
next_node_ptr builder);
                let pixel_ptr = L.build_struct_gep newnode 1 "pixel" builder in
                let pxlv = lookup px_s locals in
                ignore(L.build_store pxlv pixel_ptr builder);
                let canlv = lookup can_s locals in
                let headptr = L.build_struct_gep canlv 2 "head" builder in
                let oldhead = L.build_load headptr "oldptr" builder in
                ignore(L.build_store oldhead next_node_ptr builder);
                ignore(L.build_store newnode headptr builder); canlv
            | _ -> raise (Failure ("improper usage of shoehorn: -> append() " ^
                    (string_of_sexpr can) ^ " and " ^ (string_of_sexpr crv))))
      | SBinop((A.CanvasCircle,_) as can, op, crl) ->
          let (_,can_s) = (match (snd can) with
              SId s -> (expr builder locals can, s)
              |_-> raise(Failure "improper usage of shoehorn - canvas"))
          and (_,cl_s) = (match (snd crl) with
              SId s -> (expr builder locals crl,s)
              |_->raise(Failure "improper usage of shoehorn - circle: ->
```

```
append().circle")) in
          (match op with
              A.ShoehornCircle   ->
                (* construct new node, add it to front of list *)
                let newnode = L.build_alloca canvascirclenode_t "newnode" builder
in
                let next_node_ptr = L.build_struct_gep newnode 0 "new_circle"
builder in
                ignore(L.build_store (L.const_null (L.pointer_type
canvascirclenode_t)) next_node_ptr builder);
                let circle_ptr = L.build_struct_gep newnode 1 "circle" builder in
                let pxlv = lookup cl_s locals in
                ignore(L.build_store pxlv circle_ptr builder);
                let canlv = lookup can_s locals in
                let headptr = L.build_struct_gep canlv 2 "head" builder in
                let oldhead = L.build_load headptr "oldptr" builder in
                ignore(L.build_store oldhead next_node_ptr builder);
                ignore(L.build_store newnode headptr builder); canlv
            | _ -> raise (Failure ("improper usage of shoehornCircle with " ^
                    (string_of_sexpr can) ^ " and " ^ (string_of_sexpr crl))))
      | SBinop (e1, op, e2) ->
          let e1' = expr builder locals e1
          and e2' = expr builder locals e2 in
          (match op with
              A.Add     -> L.build_add
            | A.Sub     -> L.build_sub
            | A.Mult    -> L.build_mul
            | A.Div     -> L.build_sdiv
            | A.Mod     -> L.build_srem
            | A.And     -> L.build_and
            | A.Or      -> L.build_or
            | A.Equal   -> L.build_icmp L.Icmp.Eq
            | A.Neq     -> L.build_icmp L.Icmp.Ne
            | A.Less    -> L.build_icmp L.Icmp.Slt
            | A.Leq     -> L.build_icmp L.Icmp.Sle
            | A.Greater -> L.build_icmp L.Icmp.Sgt
            | A.Geq     -> L.build_icmp L.Icmp.Sge
            | _         -> raise (Failure "illegal binary operation")
          ) e1' e2' "tmp" builder
```

```
      | SUnop(op, ((t, _) as e)) ->
          let e' = expr builder locals e in
          (match op with
              A.Neg when t = A.Float -> L.build_fneg
            | A.Neg                  -> L.build_neg
            | A.Not                  -> L.build_not)
          e' "tmp" builder
      | SCall ("print", [e]) | SCall ("printb", [e]) ->
          L.build_call printf_func [| int_format_str ; (expr builder locals e) |]
              "printf" builder
      | SCall ("prints", [e]) ->
          L.build_call printf_func [| str_format_str ; (expr builder locals e) |]
              "printf" builder
      | SCall ("printbig", [e]) ->
          L.build_call printbig_func [| (expr builder locals e) |]
              "printbig" builder
      | SCall ("printf", [e]) ->
          L.build_call printf_func [| float_format_str ; (expr builder locals e)
|]
              "printf" builder
      | SCall ("draw", [can;name]) ->
          let can' = expr builder locals can
          and name' = expr builder locals name in
          L.build_call draw_func [| can' ; name' |]
              "draw" builder

      | SCall ("drawcircle", [cancir;name]) ->
          let cancir' = expr builder locals cancir
          and name' = expr builder locals name in
          L.build_call drawcircle_func [| cancir' ; name' |]
              "drawcircle" builder

      | SCall ("Point", [f1;f2]) ->
          let f1' = expr builder locals f1
          and f2' = expr builder locals f2 in
          L.build_call ptcons_func [| f1' ; f2' |] "Point" builder
      | SCall ("Pixel", [p1]) ->
          let p1' = expr builder locals p1 in
```

```
        L.build_call ccons_func [| p1' |] "Pixel" builder

    | SCall ("Circle", [f1;f2]) ->
        let f1' = expr builder locals f1
        and f2' = expr builder locals f2 in
        L.build_call circlecons_func [| f1' ; f2' |] "Circle" builder

    | SCall ("Canvas", [x ; y]) ->
        let x' = expr builder locals x
        and y' = expr builder locals y in
        L.build_call canvascons_func [| x' ; y' |] "Canvas" builder

    | SCall ("CanvasCircle", [x ; y]) ->
        let x' = expr builder locals x
        and y' = expr builder locals y in
        L.build_call canvascirclecons_func [| x' ; y' |] "CanvasCircle" builder

    | SCall (fname, args) ->
        let (ldev, sfd) = StringMap.find fname function_decls in
        let actuals = List.rev (List.map (fun e -> expr builder locals e)
            (List.rev args)) in
        let ret = (match sfd.styp with
            A.Void -> ""
          | _-> fname^"_ret") in
        L.build_call ldev (Array.of_list actuals) ret builder
  in

  (* LLVM insists each basic block end with exactly one "terminator"
     instruction that transfers control.  This function runs "instr builder"
     if the current block does not already have a terminator.  Used,
     e.g., to handle the "fall off the end of the function" case. *)
  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()
    | None -> ignore (instr builder) in

  (* Build the code for the given statement; return the builder for
     the statement's successor (i.e., the next instruction will be built
     after the one generated by this call) *)
```

```
    let rec stmt builder locals = function
        SBlock sl -> List.fold_left (fun (b, lv) s -> stmt b lv s) (builder,
locals) sl
      | SVDecl(ty, name) ->
          let local_var = L.build_alloca (ltype_of_typ ty) name builder in
          let locals = StringMap.add name local_var locals in
          (builder, locals)
      | SVDeclAssign(ty, name, sx) ->

          let local_var = L.build_alloca (ltype_of_typ ty) name builder in
          let locals = StringMap.add name local_var locals in
          ignore (expr builder locals (ty,SAssign(name, sx))); (builder, locals)
      | SExpr e -> ignore(expr builder locals e); (builder, locals)
      | SReturn e -> ignore(match fdecl.styp with
                              (* Special "return nothing" instr *)
                              A.Void -> L.build_ret_void builder
                              (* Build return statement *)
                            | _ -> L.build_ret (expr builder locals e) builder );
                     (builder, locals)
      | SIf (predicate, then_stmt, else_stmt) ->
           let bool_val = expr builder locals predicate in
           let merge_bb = L.append_block context "merge" the_function in
           let build_br_merge = L.build_br merge_bb in (* partial function *)
           let then_bb = L.append_block context "then" the_function in
           add_terminal (fst (stmt (L.builder_at_end context then_bb) locals
then_stmt))
           build_br_merge;
           let else_bb = L.append_block context "else" the_function in
            add_terminal (fst (stmt (L.builder_at_end context else_bb) locals
else_stmt))
           build_br_merge;
           ignore(L.build_cond_br bool_val then_bb else_bb builder);
           (L.builder_at_end context merge_bb, locals)
      | SWhile (predicate, body) ->
           let pred_bb = L.append_block context "while" the_function in
           ignore(L.build_br pred_bb builder);
           let body_bb = L.append_block context "while_body" the_function in
           add_terminal (fst (stmt (L.builder_at_end context body_bb) locals
```

```
body))
           (L.build_br pred_bb);
           let pred_builder = L.builder_at_end context pred_bb in
           let bool_val = expr pred_builder locals predicate in
           let merge_bb = L.append_block context "merge" the_function in
           ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
           (L.builder_at_end context merge_bb, locals)

      (* Implement for loops as while loops *)
      | SFor (e1, e2, e3, body) -> stmt builder locals
         ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
      in

      (* Build the code for each statement in the function *)
      let (builder, _ ) = stmt builder local_vars (SBlock fdecl.sbody) in
        (* Add a return if the last block falls off the end *)
        add_terminal builder (match fdecl.styp with
            A.Void -> L.build_ret_void
      | A.Float -> L.build_ret (L.const_float float_t 0.0)
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body functions;
  the_module
```

Sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)
(*  Authors: Adar Tulloch, Vishrut Tiwari, Winston Zhang, and Jack LaVelle *)


open Ast
```

```
type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SCharLit of char
  | SStringLit of string
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SField of string * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SNoexpr

type sstmt =
    SVDecl of typ * string
  | SVDeclAssign of typ * string * sexpr
  | SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    sbody : sstmt list;
  }

type sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SLiteral(l) -> string_of_int l
```

```
  | SFliteral(l) -> l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SCharLit(l) -> String.make 1 l
  | SStringLit(l) -> l
  | SField(e,f) -> e ^ "." ^ string_of_sexpr f
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SNoexpr -> ""  ) ^ ")"

let rec string_of_sstmt = function
    SVDecl(t, i) -> string_of_typ t ^ " " ^ i ^ "\n"
  | SVDeclAssign(t, i, e) -> string_of_typ t ^ " " ^ i ^ " = " ^ string_of_sexpr
e ^ "\n"
  | SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
  ")\n{\n" ^
  (* String.concat "" (List.map string_of_vdecl fdecl.slocals) ^ *)
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"
```

```
let string_of_sprogram (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

scanner.mll

```
(* Ocamllex scanner for See++ *)
(*     Authors: Authors: Adar Tulloch, Vishrut Tiwari, Winston Zhang, and Jack
LaVelle *)

{ open Parserseepp }

let digit = ['0' - '9']
let digits = digit+
let append = "-> append()"
let appendCircle = "-> append().circle"

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| "//"     { single lexbuf }            (* Single line comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '.'      { DOT }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MOD }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
```

```
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "break"  { BREAK }
| "continue" { CONTINUE }
| "int"    { INT }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "void"   { VOID }
| "char"   { CHAR }
| "String" { STRING }
| "Point"  { POINT }
| "Pixel"  { PIXEL }
| "Circle" { CIRCLE }
| "Canvas" { CANVAS }
| "CanvasCircle" { CANVASCIRCLE }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| append   { SHOEHORN }
| appendCircle { SHOEHORNCIRCLE}
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
| eof { EOF }
| ''' (_ as ch) ''' { CHAR_LITERAL(ch) }
| '"' ([^ '"']* as str) '"' { STRING_LITERAL(str) }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
```

```
| _    { comment lexbuf }

and single = parse
  '\n' { token lexbuf }
| _    { single lexbuf }
```

seepp.ml

```
(*Authors: Adar Tulloch, Vishrut Tiwari, Winston Zhang, and Jack LaVelle*)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./microc.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parserseepp.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
  Llvm_analysis.assert_valid_module m;
```

```
  print_string (Llvm.string_of_llmodule m)
```

semant.ml

```
open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =

  (* Verify a list of bindings has no void types or duplicate names *)
  let check_binds (kind : string) (binds : bind list) =
    List.iter (function
  (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) binds;
    let rec dups = function
        [] -> ()
      | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
    raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in

  (**** Check global variables ****)

  check_binds "global" globals;

  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no bodies *)
```

```
  let built_in_decls =
    let add_bind map (name, retyp, formlist) = StringMap.add name {
      typ = retyp;
      fname = name;
      formals = formlist;
      (* locals = []; *) body = [] } map
    in List.fold_left add_bind StringMap.empty [ ("print", Void, [(Int, "x")]);
                         ("printb", Void, [(Bool, "x")]);
                         ("printf", Void, [(Float, "x")]);
                         ("printbig", Void, [(Int, "x")]);
                         ("prints", Void, [(String, "x")]);
                         ("draw", Void, [(Canvas, "can"); (String, "filename")]);
                         ("drawcircle", Void, [(CanvasCircle, "cancir"); (String,
"filename")]);
                         ("Point", Point, [(Float, "x"); (Float, "y")]);
                         ("Pixel", Pixel, [(Point, "ep1")]);
                         ("Circle", Circle, [(Point, "ep1"); (Float, "x")]);
                         ("Canvas", Canvas, [(Float, "x"); (Float, "y")]);
                         ("CanvasCircle", CanvasCircle, [(Float, "x"); (Float,
"y")]);]

  in

  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *)
    in match fd with (* No duplicate functions or redefinitions of built-ins *)
         _ when StringMap.mem n built_in_decls -> make_err built_in_err
       | _ when StringMap.mem n map -> make_err dup_err
       | _ ->  StringMap.add n fd map
  in

  (* Collect all function names into one symbol table *)
  let function_decls = List.fold_left add_func built_in_decls functions
  in
```

```
  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  let _ = find_func "main" in (* Ensure "main" is defined *)

  let check_function func =
    (* Make sure no formals or locals are void or duplicates *)
    check_binds "formal" func.formals;
    (* check_binds "local" func.locals; *)


  (* Raise an exception if the given rvalue type cannot be assigned to
       the given lvalue type *)
  let check_assign lvaluet rvaluet err =
     if lvaluet == rvaluet then lvaluet else raise (Failure err)
    in


    (* Build initial symbol table with globals and formals *)
    let globmap = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
        StringMap.empty (globals @ func.formals)
    in

    (* Return type of a symbol from supplied symbol table *)
    let type_of_identifier locals s =
      try StringMap.find s locals
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in

    (* Return member symbol map for a particular type *)
    let member_map_of_type ty = match ty with
        Point
      | Canvas -> List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
            StringMap.empty [(Float, "x"); (Float, "y")]
      | CanvasCircle -> List.fold_left (fun m (ty, name) -> StringMap.add name ty
m)
```

```
            StringMap.empty [(Float, "x"); (Float, "y")]
      | Pixel -> List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
            StringMap.empty [(Point, "ep1")]
      | Circle -> List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
            StringMap.empty [(Point, "ep1"); (Float, "x")]
      | _ -> raise (Failure ("type " ^ string_of_typ ty ^ " does not have
members"))

     in


    (* Return a semantically-checked expression, i.e., with a type *)

    let rec expr locals = function
        Literal  l  -> (Int, SLiteral l)
      | Fliteral l  -> (Float, SFliteral l)
      | BoolLit l   -> (Bool, SBoolLit l)
      | CharLit l   -> (Char, SCharLit l)
      | StringLit l -> (String, SStringLit l)
      | Noexpr      -> (Void, SNoexpr)
      | Id s        -> (type_of_identifier locals s, SId s)
      | Assign(var, e) as ex ->
          let lt = type_of_identifier locals var
          and (rt, e') = expr locals e in
          let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
            string_of_typ rt ^ " in " ^ string_of_expr ex ^ " for identifier " ^
var
          in (check_assign lt rt err, SAssign(var, (rt, e')))
      | Field(obj, mem)  ->
          let ty = type_of_identifier locals obj in
          let memmap = member_map_of_type ty in
          let smem = match mem with
              Assign(v,e) as ex->
                   let ty = type_of_identifier memmap v in
                      (match e with
                         Fliteral _ ->
                            let lt = StringMap.find v memmap
                            and (rt, e') = expr locals e in
                            let err = "illegal assignment of object field" ^
```

```
                            string_of_typ lt ^ " = " ^
                            string_of_typ rt ^ " in " ^
                            string_of_expr ex ^ " for identifier Field." ^ v
                        in (check_assign lt rt err, SAssign(v, (rt, e')))
                    | Id s ->  (ty,SAssign(v,(ty, SId s)) )
                    | _ -> raise (Failure ("illegal member access - "
                              ^ " expression type is not a field")))
          | _ -> expr memmap mem
        in
      (fst smem, SField(obj, smem))

  | Unop(op, e) as ex ->
      let (t, e') = expr locals e in
      let ty = match op with
        Neg when t = Int || t = Float -> t
      | Not when t = Bool -> Bool
      | _ -> raise (Failure ("illegal unary operator " ^
                             string_of_uop op ^ string_of_typ t ^
                             " in " ^ string_of_expr ex))
      in (ty, SUnop(op, (t, e')))
  | Binop(e1, op, e2) as e ->
      let (t1, e1') = expr locals e1
      and (t2, e2') = expr locals e2 in
      (* All binary operators require operands of the same type *)
      let same = t1 = t2 in
      (* Determine expression type based on operator and operand types *)
      let ty = match op with
        Add | Sub | Mult | Div when same && t1 = Int   -> Int
      | Add | Sub | Mult | Div when same && t1 = Float -> Float
      | Equal | Neq            when same               -> Bool
      | Less | Leq | Greater | Geq
                when same && (t1 = Int || t1 = Float) -> Bool
      | And | Or when same && t1 = Bool -> Bool
      | Mod when same && t1 = Int -> Int
      | Shoehorn when t1 = Canvas && t2 = Pixel -> Canvas
      | ShoehornCircle when t1 = CanvasCircle && t2 = Circle -> CanvasCircle
      | _ -> raise (
    Failure ("illegal binary operator " ^
                  string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
```

```
                    string_of_typ t2 ^ " in " ^ string_of_expr e))
      in (ty, SBinop((t1, e1'), op, (t2, e2')))
  | Call(fname, args) as call ->
      let fd = find_func fname in
      let param_length = List.length fd.formals in
      if List.length args != param_length then
        raise (Failure ("expecting " ^ string_of_int param_length ^
                        " arguments in " ^ string_of_expr call))
      else let check_call (ft, _) e =
        let (et, e') = expr locals e in
        let err = "illegal argument found " ^ string_of_typ et ^
          " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
        in (check_assign ft et err, e')
      in
      let args' = List.map2 check_call fd.formals args
      in (fd.typ, SCall(fname, args'))
in

let check_bool_expr locals e =
  let (t', e') = expr locals e
  and err = "expected Boolean expression in " ^ string_of_expr e
  in if t' != Bool then raise (Failure err) else (t', e')
in

(* Return a semantically-checked statement i.e. containing sexprs *)

let rec check_stmt locals = function
  Block sl ->
    let rec check_block block_locals ssl= function
    [Return _ as s] -> ssl @ [check_stmt block_locals s]
      | Return _::_ -> raise (Failure "nothing may follow a return")
      | Block sl :: ss -> [check_stmt block_locals (Block sl)]
                          @ (check_block block_locals ssl ss)
      | s :: ss ->
        (match s with
            VDecl(t,name) ->
              (match t with
                  Void -> raise(Failure ("illegal void local "^name))
                | _ -> let block_locals = StringMap.add name t block_locals
```

```
                        in [check_stmt block_locals s] @ check_block
block_locals ssl ss)
             | VDeclAssign(t,name,e) ->
               if t == Void then raise(Failure ("illegal void local "^name) )
               else
               let sx = expr block_locals e in
               let typ =
                 if fst(sx) == t
                         then fst(sx)
                         else raise(Failure("illegal assignment")) in
               let block_locals = StringMap.add name typ block_locals in
                 [check_stmt block_locals s] @ check_block block_locals ssl ss
             | _ -> [check_stmt block_locals s] @ check_block block_locals ssl
ss)
         | []  -> ssl
       in SBlock(check_block locals [] sl)
   | VDecl(t,s) -> SVDecl(t,s)
   | VDeclAssign(_,s,e) ->
     let sx = expr locals e in
     let ty = type_of_identifier locals s in
     SVDeclAssign(ty,s,sx)
   | Expr e -> SExpr (expr locals e)
   | If(p, b1, b2) -> SIf(check_bool_expr locals p, check_stmt locals b1,
                          check_stmt locals b2)
   | For(e1, e2, e3, st) ->
       SFor(expr locals e1, check_bool_expr locals e2, expr locals e3,
            check_stmt locals st)
   | While(p, s) -> SWhile(check_bool_expr locals p, check_stmt locals s)
   | Return e -> let (t, e') = expr locals e in
       if t = func.typ then SReturn (t, e')
       else raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
      string_of_typ func.typ ^ " in " ^ string_of_expr e))
   in (* body of check_function *)
   { styp = func.typ;
     sfname = func.fname;
     sformals = func.formals;
     sbody = match check_stmt globmap (Block func.body) with
        SBlock(sl) -> sl
     | _ -> raise (Failure ("internal error: block didn't become a block?"))
```

```
    }
  in (globals, List.map check_function functions)
```

parserseepp.mly

```
/* Ocamlyacc parser for See++ */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES MOD DIVIDE ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR
%token DOT PIPE SHOEHORN SHOEHORNCIRCLE
%token RETURN IF /*ELIF*/ ELSE FOR WHILE INT BOOL FLOAT VOID
%token BREAK CONTINUE
%token CHAR STRING POINT PIXEL CIRCLE CANVAS CANVASCIRCLE
%token <int> LITERAL
%token <bool> BLIT
%token <char> CHAR_LITERAL
%token <string> STRING_LITERAL
%token <string> ID FLIT
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
```

```
%left MOD
%left PIPE SHOEHORN SHOEHORNCIRCLE
%left DOT
%right NOT NEG

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { ([], [])               }
 | decls vdecl { (($2 :: fst $1), snd $1) }
 | decls fdecl { (fst $1, ($2 :: snd $1)) }

fdecl:
  typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE {
    { typ = $1;
      fname = $2;
      formals = List.rev $4;
      body = List.rev $7 }
    }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { $1 }

formal_list:
    typ ID                   { [($1,$2)]     }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
    INT       { Int    }
  | BOOL      { Bool   }
  | FLOAT     { Float  }
  | VOID      { Void   }
  | CHAR      { Char   }
  | STRING    { String }
  | POINT     { Point  }
```

```
  | PIXEL     { Pixel  }
  | CIRCLE    { Circle }
  | CANVAS    { Canvas }
  | CANVASCIRCLE { CanvasCircle}

vdecl:
    typ ID SEMI { ($1, $2) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

vdecl_stmt:
    typ ID SEMI { VDecl($1,$2)}
  | typ ID ASSIGN expr SEMI { VDeclAssign($1, $2, $4) }

stmt:
    expr SEMI                               { Expr $1               }
  | vdecl_stmt                              { $1                    }
  | RETURN expr_opt SEMI                    { Return $2             }
  | LBRACE stmt_list RBRACE                 { Block(List.rev $2)    }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)        }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                            { For($3, $5, $7, $9)   }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1)            }
  | FLIT             { Fliteral($1)           }
  | BLIT             { BoolLit($1)            }
  | ID               { Id($1)                 }
  | CHAR_LITERAL     { CharLit($1)            }
  | STRING_LITERAL   { StringLit($1)          }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
```

```
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $3)   }
  | expr MOD    expr { Binop($1, Mod,   $3)   }
  | expr EQ     expr { Binop($1, Equal, $3)   }
  | expr NEQ    expr { Binop($1, Neq,   $3)   }
  | expr LT     expr { Binop($1, Less,  $3)   }
  | expr LEQ    expr { Binop($1, Leq,   $3)   }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3)   }
  | expr AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | expr SHOEHORN expr { Binop($1, Shoehorn, $3)  }
  | expr SHOEHORNCIRCLE expr { Binop($1, ShoehornCircle, $3) }
  | ID   DOT    expr                  { Field($1, $3)               }
  | MINUS expr %prec NOT              { Unop(Neg, $2)               }
  | NOT expr                          { Unop(Not, $2)               }
  | ID ASSIGN expr                    { Assign($1, $3)              }
  | ID LPAREN args_opt RPAREN         { Call($1, $3)                }
  | typ LPAREN args_opt RPAREN        { Call((string_of_typ $1), $3) }
  | LPAREN expr RPAREN                { $2 }

args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

args_list:
    expr                    { [$1] }
  | args_list COMMA expr    { $3 :: $1 }
```