PyLit

Final Report

Ryan LoPrete (rjl2172)

December 19, 2018

Table of Contents:

1 Introduction

1.1 Context

Although Python is widely considered a scripting language, it has grown into a general-purpose tool with an active and supportive community. Given its versatility and extensive library support, developers even at the enterprise level are leveraging the power of Python for web development, building APIs, big data, and machine learning. Much less emphasis is placed on conventional syntax, giving programmers from all walks of life the ability to quickly pick up the language and make it immediately purposeful.

In the same spirit of what makes Python so special was the inspiration for a language of similar stature, PyLit.

1.2 Aim and Motivations

PyLit aims to provide a simple, lightweight implementation of the increasingly common, general-purpose language, Python. Written in OCaml, the PyLit compiler enables programmers to quickly generate code in a similar syntax to that of Python. Users will notice that creating algorithms in PyLit is refreshingly easy, and there a few opportunities to introduce bugs. A key difference between the two languages is that PyLit does not support type inference. The motivation to create PyLit was promoted by the idea that someday a language like this could be extended to support a plethora of library packages on the same order of magnitude as Python. PyLit's built-in functions could also be extended to compete with the offerings of similar languages.

2 Language Tutorial

2.1 Environment Setup

PyLit was developed in OCaml. In order to use the compiler, OCaml must be installed properly. The below steps are for installation on Ubuntu 16.04. See the README for additional instructions.

Run the below commands to install the necessary packages. LLVM 3.8 is the default under Ubuntu 16.04, so a matching installation of the OCaml LLVM bindings are needed.

```
$ sudo apt install ocaml llvm llvm-runtime m4 opam
$ opam init
$ opam install llvm.3.8
$ eval `opam config env`
```

Following these commands, install clang++ in order to convert the assembly that the compiler generates into executable files.

Once installation is complete, run the testall.sh script to execute the PyLit regression test suite. All tests should pass if setup was successful.

2.2 Sample Program

After installation of the compiler is complete, users are ready to start coding. Let's create a simple "Hello World" program to begin. Create a file named HelloWorld.pyl with the code below.

```
1 # HelloWorld.pyl
2 def void main():
3   prints("Hello World!");
4 end
```

To execute the program, run the below commands from the terminal.

```
$ ./pylit.native < ./HelloWorld.pyl > HelloWorld.bc
$ clang -o HelloWorld HelloWorld.bc
$ ./HelloWorld
```

You should see "Hello World!" as the output!

3 Language Manual

Types

Types are explicitly declared in PyLit. Below are the five basic data types.

string: String literals contain text consisting of ASCII characters between double quotes.

```
let str = ([' '-'!' '#'-'[' ']'-'~']| '\\' ['\\' '"' 'n' 't'
'r'])*
```

int: Integer literals are any whole number without a decimal or fractional portion.

```
let digit = ['0'-'9']
```

float: Floats are used to represent decimal numbers or those containing exponents. For example, 0.5 and 1E7 are considered floats. The matching rule for floats is given below.

```
let exponent = ['e' 'E'] ['+' '-']? ['0'-'9'] (['0'-'9'] | '_')*
let float =
   ('-'? ['0'-'9'] (['0'-'9'] | '_')*)? (('.' (['0'-'9'] | '_')*
(exponent)?) | exponent)
```

none: None types do not provide a result value to its caller.

bool: Booleans represent values which are either logical true / false. These are reserved keywords

Examples:

```
var = 1          # int
var = 1.5        # float
var = "PyLit"    # string
var = true       # bool
```

Lexical Conventions

Statement Termination

Statements should be terminated with the semicolon ';'. This character tells PyLit to evaluate the statement preceding that character.

Comments

Single line comments are allowed in PyLit and are denoted by the '#' character.

```
# This sentence is a comment

x = 10; # Comments can also appear after statements
```

Whitespace

Spaces, horizontal tab, carriage return, and new line are ignored by PyLit. Tokens can be separated by any of the above characters without affecting code execution.

```
whitespace = [' ' '\t' '\r' '\n']
```

Identifiers

Identifiers are used to represent variables. An identifier can consist of the below characters, but cannot be one of PyLit's reserved keywords.

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

Example valid identifiers:

```
var         first_name
Option_1    var2
```

Example invalid identifiers:

```
var-2       5_a
if          *a
```

Keywords

PyLit consists of a number of keywords that hold special meaning. These are reserved and cannot be used as variable declarations.

```
def, end, if, else, for, while, return, int, bool, float, none,
str, true, false
```

Separators

The following list contains characters that separate tokens, other than whitespace. They are used in function declarations or defining blocks, for example.

```
 '('          { LP }
 ')'          { RP }
 '['          { LSB }
 ']'          { RSB }
```

```
'{'             { LBRACE }
'}'             { RBRACE }
','             { COMMA }
';'             { SEMICOLON }
':'             { COLON }
```

Operators

PyLit offers unary and binary operators. The two unary operators are negation via the '-' token, as well as '!' for logical negation.

```
'+'             { PLUS }
'-'             { MINUS }
'*'             { MULTIPLY }
'/'             { DIVIDE }
'%'             { MOD }
'<'             { LT }
'>'             { GT }
'='             { ASSIGNMENT }
'!'             { NOT }
"<="            { LTE }
">="            { GTE }
"=="            { EQUALS }
"!="            { NEQUAL }
"&&"            { AND }
"||"            { OR }
```

Arithmetic Operations

Basic arithmetic can be calculated on floats and integers. Two operands must have matching types or an error will be thrown. The five arithmetic operations as well as unary negation are shown below.

Operation	Example
Addition	`1 + 1;  # 2`
Subtraction	`5 - 4;  # 1`
Multiplication	`3 * 2;  # 6`
Division	`10 / 5;  # 2`
Modulus	`10 % 2;  # 0`
Unary Minus	`--5;  # 5`

Relational Operators

Relational operations can be performed on types float, int, or string. The operands must be of the same type. For comparison of string types, alphabetical ordering is used and

returns a Boolean type (true or false). Variables can also be compared if the identifiers specify similar types. Examples of the relational operators are shown below.

Less than:

```
10 < 20;                # evaluates to true
```

Greater than:

```
20.0 > 10.0;            # evaluates to true
```

Less than or equal:

```
"PyLit" >= "PyLit";     # evaluates to true
```

Greater than or equal:

```
"PyLit" > "PyLit";      # evaluates to false
```

Variable comparison:

```
a = 5;
b = 3;
b > a;                  # evaluates to false
```

String Operators

PyLit offers one string operations for concatenation, given by the "+" token. Both operands must be string literals or variables that identify string literals.

```
"PyLit" + " is great"; # evaluates to "PyLit is great"
```

Comparison Operators

Comparison operators include equals "==" and not equal "!=". Any primitive operands can be compared but they must be of the same type. Strings are equal if they contain the same alphabetical characters in the same order, while lists and dictionaries are equal if they contain the same values or key/value pairs in the same order.

```
"PyLit" == "PyLit"      # evaluates to true
5.0 == 5.0              # evaluates to true
5 == 5.0                # evaluates to false
```

Logical Operators

Three logical operators are offered in PyLit: logical "and", logical "or", and logical "not". Any expression that evaluates to a Boolean literal can use these operations. The returned type is also a boolean. Logical "and" / "or" take two operands, while "not" is unary.

```
true and true    # evaluates to true
true and false   # evaluates to false
true or false    # evaluates to true
false or false   # evaluates to false
!true            # evaluates to false
```

Operator Precedence

Precedence	Operator
Highest	! (unary)
	* , / , %
	+ , -
	<=, >=, <, >, ==, !=
	&&
	||
Lowest	Assign (=)

All operations are left associative in PyLit except for ‘assign’ and ‘not’, which are right associative.

Functions

PyLit supports function declaration, function calls, control flow, global variables, and recursion. Every function has an argument that takes 0 or more variables, surrounded by parentheses. Variables passed into called functions must match types used when declaring the function See Section 8 for a variety of examples including for/while loops, printing functions, and arithmetic.

Scoping

PyLit is statically scoped in a similar way to that of the C language. Rather than using brackets to define the beginning and ending of blocks, PyLit uses colons and the ‘end’ keyword like Python. Local variables are declared and accessed inside a function block while global variables can be accessed anywhere within the program. Global variables will be overwritten when the same variable is declared within a function block. Scoping example shown below.

```
1 int i;
2
3 def none main():
4   str i;  #Global i is out of scope
5   i = "Hello";
6   prints(i);
7 end
```

output: "Hello"

Sample Algorithms

```
 1 # factorial.pyl
 2 def int factorial(int x):
 3   if (x < 2):
 4     return 1;
 5   end
 6   return x * factorial(x-1);
 7 end
 8
 9 def int main():
10   print(factorial(5));
11 end
```

output: 120

```
 1 def none isEven(int number):
 2   str result;
 3   if (number%2 != 0):
 4     result = "Number is odd!";
 5   end
 6   else:
 7     result = "Number is even!";
 8   end
 9   prints(result);
10 end
11
12 def int main():
13   isEven(10);
14 end
```

output: Number is even!

4 Project Plan

4.1 Planning Process

Throughout the project an iterative process was used to ensure the timely delivery of multiple milestones. Milestones were chosen based off of an initial dive into project requirements as well as by following the due dates for major deliverables as defined by Prof. Edwards. It is difficult to determine from the get-go exactly what challenged will be faced and the compiler is coded, so short-term goals were often introduced as a subset of the key benchmarks.

4.2 Specification Process

As mentioned in the introduction, initial specifications for PyLit were inspired by the Python language. Creating a derivative of Python is a vast undertaking, but for the scope of this project, only a small selection of features was implemented. Creating a project proposal helped flush out which features would attempt to be implemented. Since the project proposal was generated near the start of the class, there remained nearly an entire semester's worth of learning to be completed. This meant that all the knowledge of how to implement particular compiler features was not known at the time of creating the proposal. Naturally, features were scaled back over time that proved to be outside the scope of the class, such as type inference.

4.3 Development Process

Development followed the flow of an end-to-end compiler, starting with the scanner. Following this was the parser, AST, semantic checker, SAST, and the code generator. As modifications were needed throughout development and testing, some translator components were revisited for editing. Having a fully functioning parser without the presence of shift/reduce conflicts was a class defined deliverable that helped keep the process on track. Additionally, some helper components were created as needed such an imported library of custom C functions to be linked within the compiler.

A critical part of creating this translator was to incorporate time for learning OCaml. A project requirement was to strictly use this language from end-to-end, excluding the linking of any external libraries. Class homework and independent study are to be thanked for overcoming this learning curve.

4.4 Testing Process

PyLit has fairly large regression test suite consisting of over 70 test cases. As features were implemented, test programs were written and added to the regression suite. After a feature was fully implemented, the regression suite was run to ensure previously functional aspects of the translator remained intact. If there were test failures, troubleshooting would ensue to comb over the most recently added code in search of bugs. At times, smaller, more individualistic test programs were created as a result of failed regression cases to pinpoint the root cause of a bug. It can be difficult to determine at which point in the compilation process causes a test failure, but over time this is learned as an art.

4.5 Programming Style Guide

To ensure consistency and readability, the below standards were followed:

- Single tabs for indentation throughout each program file
- Align function arrows (->) when multiple are used
- Lines should not exceed 80 characters

4.6 Project Timeline

September 19	Submitted *Project Proposal*
September 24	Configured development environment
September 28	Completed lexical analyzer
October 12	Submitted *Language Reference Manual* and Parser
November 14	Submitted *Hello World*
November 30	Semantics and AST complete
December 7	Code generation complete
December 14	Regression testing complete
December 19	Submission of Final Project Report

4.7 Roles and Responsibilities

Given that this project was completed independently, I assumed the roles of Tester, System Architect, Project Manager, and Language Guru. Bringing the compiler from conception to completion was at times made more challenging without the help of teammates to exchange ideas with or to get a fresh pair of eyes when debugging. Meeting regularly with the TAs and Prof. Edwards became a critical part of resolving bottlenecks.

4.8 Software Development Environment

The following packages were used during development:

- Ubuntu 18.04.1
- OCaml version 4.05.0 (ocamlyacc and ocamllex extensions used)
- LLVM version 6.0
- Opam version 1.2.2
- Clang version 6.0.0
- GCC 7.3.0 (for building C output)
- Atom IDE with ocaml-merlin plugin

4.9 Project Log

As mentioned, the project was completed independently so there wasn't a need for a version control system. All code additions and edits were saved locally.

5 Architectural Design

5.1 Block Diagram

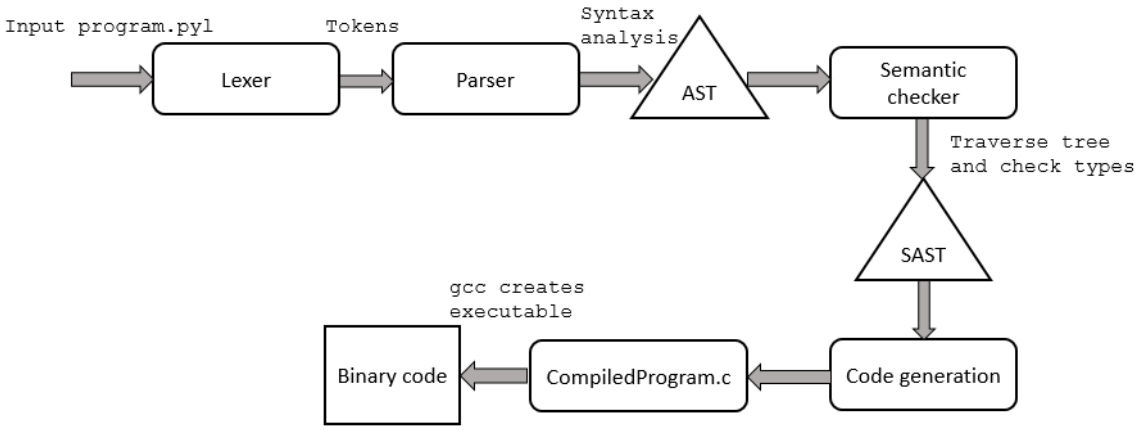

Figure 1. *Architecture of PyLit Translator*

5.2 Component Interactions

Lexer

The lexer leverages ocamllex to produce a lexical analyzer from a set of regular expressions with associated semantic actions. It takes a PyLit source file as it's input and generates tokens based on acceptable values defined in the scanner source code.

Parser / AST

The parser takes in tokens generated from the lexer. Using ocamlyacc, a parser is produced from the PyLit production rules specified in the source code. An abstract syntax tree is constructed given the datatypes defined in the AST source code. If this stage passes, it means the program is syntactically, but not necessarily semantically, correct.

Semantic Checker

The semantic module recursively traverses the AST and converts it into a semantically checked abstract syntax tree (SAST). The SAST adds type annotations to expressions, but is the same as the AST otherwise. The SAST is needed because the code generator needs type information to perform correctly. If this fails, there is an error in the PyLit program.

Code Generator

After the AST is checked for semantic accuracy, the code generator converts the AST into LLVM IR. This intermediate representation is assembly language that can be used for code generation of Intel X86 binary, ARM, etc.

C Library

A source file was created called functions.c that contains numerous C functions used in linking C code to code generated from LLVM. These C functions help in defining the behavior for PyLit operations such as string concatenation.

6 Test Plan

6.1 Source and Target Language Programs

Below are some examples of PyLit source programs and the respective target language program that is generated (LLVM IR).

Hello World:

PyLit source code:

```
1 #HelloWorld.pyl
2 def none main():
3   prints("Hello World!");
4 end
```

Generated LLVM:

```
1 ; ModuleID = 'PyLit'
2 source_filename = "PyLit"
3
4 @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
5 @fmt.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
6 @fmt.2 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
7 @str = private unnamed_addr constant [13 x i8] c"Hello World!\00"
8
9 declare i32 @printf(i8*, ...)
10
11 declare i32 @printbig(i32)
12
13 declare i8* @string_concat(i8*, i8*)
14
15 define void @main() {
16 entry:
17   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt.1, i32 0, i32 0), i8* getelementptr inbounds ([13 x i8],
[13 x i8]* @str, i32 0, i32 0))
18   ret void
19 }
```

Addition:

PyLit source code:

```
 1 #Addition.pyl
 2 def int add(int a, int b):
 3   return a + b;
 4 end
 5
 6 def int main():
 7   int a;
 8   a = add(7, 2);
 9   print(a);
10 end
```

Generated LLVM:

```
 1 ; ModuleID = 'PyLit'
 2 source_filename = "PyLit"
 3
 4 @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
 5 @fmt.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
 6 @fmt.2 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
 7 @fmt.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
 8 @fmt.4 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
 9 @fmt.5 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
10
11 declare i32 @printf(i8*, ...)
12
13 declare i32 @printbig(i32)
14
15 declare i8* @string_concat(i8*, i8*)
16
17 define i32 @main() {
18 entry:
19   %a = alloca i32
20   %add_result = call i32 @add(i32 7, i32 2)
21   store i32 %add_result, i32* %a
22   %a1 = load i32, i32* %a
23   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %a1)
24   ret i32 0
25 }
26
27 define i32 @add(i32 %a, i32 %b) {
28 entry:
29   %a1 = alloca i32
30   store i32 %a, i32* %a1
31   %b2 = alloca i32
32   store i32 %b, i32* %b2
33   %a3 = load i32, i32* %a1
34   %b4 = load i32, i32* %b2
35   %tmp = add i32 %a3, %b4
36   ret i32 %tmp
37 }
```

6.2 **Test Suite**

When creating a compiler, it is vital to maintain a robust testing framework from the start. If code is not tested, it is safe to consider it broken. As new features are added to a compiler, inevitably bugs and side-effects will alter the previously expected behavior. For this reason, PyLit's features are tested end-to-end with not only test programs that will pass, but with programs that will fail compilation. Considering multiple cases that cause the compiler to fail results in a more reliable translator, and helps flush out the expected behavior in all scenarios for both the developer and users.

See section 8.9 of the Appendix for a complete listing of the regression test cases.

6.3 **Automation**

As the compiler grew, automation became necessary to run through the test cases efficiently for quick discovery of failures. To accomplish this, a shell script is used that executes each test case sequentially and compares the output of the program to the expected result defined in the .err and .out files of the test suite. If there is a discrepancy between them, that particular test fails, and the script moves on to the next.

To run the regression test suite, enter the below commands in the terminal.

```
$ make pylit.native
$ ./testall.sh
```

Passing tests will appear with an 'OK' as shown below.

```
test-add1...OK
test-arith1...OK
test-arith2...OK
test-arith3...OK
```

7 Lessons Learned

7.1 Language Evolution

Throughout the semester, PyLit evolved to accommodate time constraints and complexity hurdles. A key difference seen between PyLit and Python is that PyLit does not offer dynamic type inference. After investigating how compilers achieve type inference such as with the Hindley-Milner system, it quickly became clear that this feature would have to be descoped. With a team of developers and prioritized time, type inference likely could have been realized in a semester's time.

7.2 Important Learnings

Building a compiler is not a trivial task. There are complex moving parts and two audiences to consider (the developer and the user) when designing the language. Having gone through the process of creating a very stripped-down version of a hearty language such as Python, one can truly appreciate the design considerations and time developers dedicate to creating a multi-purpose language. It also became apparent that the more time spent struggling through low level code to achieve robust features, the more accommodating a language becomes to end users. It is for this reason that languages like C, Java, and Python have withstood the test of time and will likely continue to be used for decades to come.

The concepts learned during class can be directly applied to coding the compiler. It's refreshing and more interesting when theory meets practice in such a harmonious way. Furthermore, the techniques learned in class don't only apply to compilers. They have broad application.

7.3 Advice for Future Teams

Start early. Whether you are an individual contributor or on a team of students undertaking compiler design, carve out enough time to code carefully and understand how / why your code is working or not working. The learning curve of coding a compiler in OCaml can only be overcome through lots of time.

8 Appendix

8.1 scanner.mll

```
(* Ocamllex scanner for PyLit *)

{
  open Pylitparse

  let unescape s =
      Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
 }

let digit = ['0' - '9']
let digits = digit+
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let string = '"' ( (ascii | escape)* as s) '"'

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| '#'     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LSQUARE }
| ']'      { RSQUARE }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MODULUS }
| '='      { ASSIGN }
| '\n'     { EOL }
| "=="     { EQ }
| "!="     { NEQ }
| "<"      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "def"    { DEF }
| "end"    { END }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
```

```
| "float"  { FLOAT }
| "none"   { NONE }
| "str"    { STRING }
| "list"   { LIST }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| 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) }
| string            { STRING_LITERAL( (unescape s) ) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

8.2 pylitparse.mly

```
/* Ocamlyacc parser for PyLit */

%{
open Ast
%}

%token SEMI COLON LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA PLUS MINUS TI
MES DIVIDE MODULUS ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR EOL
%token DEF END RETURN IF ELSE FOR WHILE INT BOOL FLOAT LIST NONE STRING
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT STRING_LITERAL
%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 MODULUS
%right NOT

%%

program:
  decls EOF { $1 }

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

fdecl:
   DEF typ ID LPAREN formals_opt RPAREN COLON vdecl_list stmt_list END
```

```
41.      { { typ = $2;
42.      fname = $3;
43.      formals = List.rev $5;
44.      locals = List.rev $8;
45.      body = List.rev $9 } }
46.
47. formals_opt:
48.     /* nothing */ { [] }
49.   | formal_list   { $1 }
50.
51. formal_list:
52.     typ ID                   { [($1,$2)]     }
53.   | formal_list COMMA typ ID { ($3,$4) :: $1 }
54.
55. typ:
56.     INT   { Int   }
57.   | BOOL  { Bool  }
58.   | FLOAT { Float }
59.   | NONE  { None  }
60.   | STRING { String }
61.   | LIST LSQUARE typ RSQUARE { List($3) }
62.
63. vdecl_list:
64.     /* nothing */    { [] }
65.   | vdecl_list vdecl { $2 :: $1 }
66.
67. vdecl:
68.    typ ID SEMI { ($1, $2) }
69.
70. stmt_list:
71.     /* nothing */  { [] }
72.   | stmt_list stmt { $2 :: $1 }
73.
74. stmt:
75.     expr SEMI                                { Expr $1               }
76.   | RETURN expr_opt SEMI                     { Return $2             }
77.   | COLON stmt_list END                      { Block(List.rev $2)    }
78.   | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
79.   | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)        }
80.   | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
81.                                              { For($3, $5, $7, $9)   }
82.   | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }
83.
84. expr_opt:
85.     /* nothing */ { Noexpr }
86.   | expr          { $1 }
87.
88. expr:
89.     LITERAL          { Literal($1)            }
90.   | FLIT               { Fliteral($1)           }
91.   | BLIT             { BoolLit($1)            }
92.   | STRING_LITERAL   { StringLit($1) }
93.   | ID               { Id($1)                 }
94.   | expr PLUS   expr { Binop($1, Add,   $3)   }
95.   | expr MINUS  expr { Binop($1, Sub,   $3)   }
96.   | expr TIMES  expr { Binop($1, Mult,  $3)   }
97.   | expr DIVIDE expr { Binop($1, Div,   $3)   }
98.   | expr MODULUS expr { Binop($1, Mod,   $3)   }
99.   | expr EQ     expr { Binop($1, Equal, $3)   }
100.    | expr NEQ    expr { Binop($1, Neq,   $3)   }
101.    | expr LT     expr { Binop($1, Less,  $3)   }
```

```
102.    | expr LEQ    expr { Binop($1, Leq,   $3)   }
103.    | expr GT     expr { Binop($1, Greater, $3) }
104.    | expr GEQ    expr { Binop($1, Geq,   $3)   }
105.    | expr AND    expr { Binop($1, And,   $3)   }
106.    | expr OR     expr { Binop($1, Or,    $3)   }
107.    | MINUS expr %prec NOT { Unop(Neg, $2)      }
108.    | NOT expr         { Unop(Not, $2)          }
109.    | ID ASSIGN expr   { Assign($1, $3)         }
110.    | ID LPAREN args_opt RPAREN { Call($1, $3)  }
111.    | LPAREN expr RPAREN { $2                   }
112.
113.  args_opt:
114.      /* nothing */ { [] }
115.    | args_list  { List.rev $1 }
116.
117.  args_list:
118.      expr                    { [$1] }
119.    | args_list COMMA expr { $3 :: $1 }
```

8.3 ast.ml

```
1.  (* Abstract Syntax Tree *)
2.
3.  type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater | Ge
    q |
4.            And | Or
5.
6.  type uop = Neg | Not
7.
8.  type typ = Int | Bool | Float | None | String | List of typ
9.
10. type bind = typ * string
11.
12. type expr =
13.     Literal of int
14.   | Fliteral of string
15.   | BoolLit of bool
16.   | Id of string
17.   | StringLit of string
18.   | Binop of expr * op * expr
19.   | Unop of uop * expr
20.   | Assign of string * expr
21.   | Call of string * expr list
22.   (* | Seq of expr list *)
23.   | Noexpr
24.
25. type stmt =
26.     Block of stmt list
27.   | Expr of expr
28.   | Return of expr
29.   | If of expr * stmt * stmt
30.   | For of expr * expr * expr * stmt
31.   | While of expr * stmt
32.
33. type func_decl = {
34.     typ : typ;
35.     fname : string;
36.     formals : bind list;
37.     locals : bind list;
38.     body : stmt list;
```

```
39.   }
40.
41. type program = bind list * func_decl list
42.
43. (* Pretty-printing functions *)
44.
45. let string_of_op = function
46.     Add -> "+"
47.   | Sub -> "-"
48.   | Mult -> "*"
49.   | Div -> "/"
50.   | Mod -> "%"
51.   | Equal -> "=="
52.   | Neq -> "!="
53.   | Less -> "<"
54.   | Leq -> "<="
55.   | Greater -> ">"
56.   | Geq -> ">="
57.   | And -> "&&"
58.   | Or -> "||"
59.
60. let string_of_uop = function
61.     Neg -> "-"
62.   | Not -> "!"
63.
64. let rec string_of_expr = function
65.     Literal(l) -> string_of_int l
66.   | Fliteral(l) -> l
67.   | StringLit s -> "\"" ^ s ^ "\""
68.   | BoolLit(true) -> "true"
69.   | BoolLit(false) -> "false"
70.   | Id(s) -> s
71.   | Binop(e1, o, e2) ->
72.       string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
73.   | Unop(o, e) -> string_of_uop o ^ string_of_expr e
74.   | Assign(v, e) -> v ^ " = " ^ string_of_expr e
75.   | Call(f, el) ->
76.     f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
77.   (* | Seq(a) -> string_of_list a *)
78.   | Noexpr -> ""
79.   and string_of_list = function
80.       l -> "[" ^ (string_of_seq l) ^ "]"
81.   and string_of_seq = function
82.       x :: y :: a -> string_of_expr x ^ ", " ^ string_of_seq (y :: a)
83.     | x :: _ -> string_of_expr x
84.     | [] -> ""
85.
86. let rec string_of_stmt = function
87.     Block(stmts) ->
88.       "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
89.   | Expr(expr) -> string_of_expr expr ^ ";\n";
90.   | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
91.   | If(e, s, Block([])) -
    > "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
92.   | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
93.       string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
94.   | For(e1, e2, e3, s) ->
95.       "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
96.       string_of_expr e3  ^ ") " ^ string_of_stmt s
97.   | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
98.
```

```
let rec string_of_typ = function
            Int -> "int"
          | Bool -> "bool"
          | Float -> "float"
          | None -> "none"
          | String -> "str"
          | List(t) -> "List(" ^ string_of_typ t ^ ")"

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

       let string_of_fdecl fdecl =
         "def " ^ 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" ^ "end"

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

8.4 sast.ml

```
(* Semantically-checked Abstract Syntax Tree *)

open Ast

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

type sstmt =
    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;
    slocals : 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
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SFliteral(l) -> l
  | SStringLit(s) -> s
  | 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
    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 =
  "def " ^ 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)
```

8.5 **semant.ml**

```
(* Semantic checking for the PyLit compiler *)

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) =
```

```
14.
15.   (* Verify a list of bindings has no none types or duplicate names *)
16.   let check_binds (kind : string) (binds : bind list) =
17.     List.iter (function
18.     (None, b) -> raise (Failure ("illegal none " ^ kind ^ " " ^ b))
19.       | _ -> ()) binds;
20.     let rec dups = function
21.         [] -> ()
22.       | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
23.         raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
24.       | _ :: t -> dups t
25.     in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
26.   in
27.
28.   (**** Check global variables ****)
29.
30.   check_binds "global" globals;
31.
32.   (**** Check functions ****)
33.
34.   (* Collect function declarations for built-in functions: no bodies *)
35.   let built_in_decls =
36.     let add_bind map (name, ty) = StringMap.add name {
37.       typ = None;
38.       fname = name;
39.       formals = [(ty, "x")];
40.       locals = []; body = [] } map
41.     in List.fold_left add_bind StringMap.empty [ ("print", Int);
42.                                          ("printb", Bool);
43.                                          ("printf", Float);
44.                                    ("prints", String);
45.                                          ("printbig", Int) ]
46.   in
47.
48.   (* Add function name to symbol table *)
49.   let add_func map fd =
50.     let built_in_err = "function " ^ fd.fname ^ " may not be defined"
51.     and dup_err = "duplicate function " ^ fd.fname
52.     and make_err er = raise (Failure er)
53.     and n = fd.fname (* Name of the function *)
54.     in match fd with (* No duplicate functions or redefinitions of built-
    ins *)
55.           _ when StringMap.mem n built_in_decls -> make_err built_in_err
56.         | _ when StringMap.mem n map -> make_err dup_err
57.         | _ ->  StringMap.add n fd map
58.   in
59.
60.   (* Collect all function names into one symbol table *)
61.   let function_decls = List.fold_left add_func built_in_decls functions
62.   in
63.
64.   (* Return a function from our symbol table *)
65.   let find_func s =
66.     try StringMap.find s function_decls
67.     with Not_found -> raise (Failure ("unrecognized function " ^ s))
68.   in
69.
70.   let _ = find_func "main" in (* Ensure "main" is defined *)
71.
72.   let check_function func =
73.     (* Make sure no formals or locals are none or duplicates *)
```

```
74.     check_binds "formal" func.formals;
75.     check_binds "local" func.locals;
76.
77.     (* Raise an exception if the given rvalue type cannot be assigned to
78.        the given lvalue type *)
79.     let check_assign lvaluet rvaluet err =
80.        if lvaluet = rvaluet then lvaluet else raise (Failure err)
81.     in
82.
83.     (* Build local symbol table of variables for this function *)
84.     let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
85.                     StringMap.empty (globals @ func.formals @ func.locals )
86.     in
87.
88.     (* Return a variable from our local symbol table *)
89.     let type_of_identifier s =
90.       try StringMap.find s symbols
91.       with Not_found -> raise (Failure ("undeclared identifier " ^ s))
92.     in
93.
94.     (* Return a semantically-checked expression, i.e., with a type *)
95.     let rec expr = function
96.         Literal  l -> (Int, SLiteral l)
97.       | Fliteral l -> (Float, SFliteral l)
98.       | BoolLit l  -> (Bool, SBoolLit l)
99.       | Noexpr     -> (None, SNoexpr)
100.              | StringLit s -> (String, SStringLit s)
101.              | Id s       -> (type_of_identifier s, SId s)
102.              | Assign(var, e) as ex ->
103.                  let lt = type_of_identifier var
104.                  and (rt, e') = expr e in
105.                  let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
106.                    string_of_typ rt ^ " in " ^ string_of_expr ex
107.                  in (check_assign lt rt err, SAssign(var, (rt, e')))
108.              | Unop(op, e) as ex ->
109.                  let (t, e') = expr e in
110.                  let ty = match op with
111.                    Neg when t = Int || t = Float -> t
112.                  | Not when t = Bool -> Bool
113.                  | _ -> raise (Failure ("illegal unary operator " ^
114.                                         string_of_uop op ^ string_of_typ t ^
115.                                         " in " ^ string_of_expr ex))
116.                  in (ty, SUnop(op, (t, e')))
117.              | Binop(e1, op, e2) as e ->
118.                  let (t1, e1') = expr e1
119.                  and (t2, e2') = expr e2 in
120.                  (* All binary operators require operands of the same type *)
121.                  let same = t1 = t2 in
122.                  (* Determine expression type based on operator and operand typ
    es *)
123.                  let ty = match op with
124.                      Add | Sub | Mult | Div | Mod when same && t1 = Int   -
    > Int
125.                    | Add | Sub | Mult | Div | Mod when same && t1 = Float -
    > Float
126.                    | Add                           when same && t1 = String -
    > String
127.                  | Equal | Neq            when same               -> Bool
128.                  | Less | Leq | Greater | Geq
129.                             when same && (t1 = Int || t1 = Float) -> Bool
130.                  | And | Or when same && t1 = Bool -> Bool
```

```
131.                 | _ -> raise (
132.                 Failure ("illegal binary operator " ^
133.                               string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
134.                               string_of_typ t2 ^ " in " ^ string_of_expr e))
135.                 in (ty, SBinop((t1, e1'), op, (t2, e2')))
136.               | Call(fname, args) as call ->
137.                 let fd = find_func fname in
138.                 let param_length = List.length fd.formals in
139.                 if List.length args != param_length then
140.                   raise (Failure ("expecting " ^ string_of_int param_length ^
141.                                   " arguments in " ^ string_of_expr call))
142.                 else let check_call (ft, _) e =
143.                   let (et, e') = expr e in
144.                   let err = "illegal argument found " ^ string_of_typ et ^
145.                     " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr
    e
146.                   in (check_assign ft et err, e')
147.                 in
148.                 let args' = List.map2 check_call fd.formals args
149.                 in (fd.typ, SCall(fname, args'))
150.           in
151.
152.           let check_bool_expr e =
153.             let (t', e') = expr e
154.             and err = "expected Boolean expression in " ^ string_of_expr e
155.             in if t' != Bool then raise (Failure err) else (t', e')
156.           in
157.
158.           (* Return a semantically-
    checked statement i.e. containing sexprs *)
159.           let rec check_stmt = function
160.               Expr e -> SExpr (expr e)
161.             | If(p, b1, b2) -
    > SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
162.             | For(e1, e2, e3, st) ->
163.             SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
164.             | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
165.             | Return e -> let (t, e') = expr e in
166.               if t = func.typ then SReturn (t, e')
167.               else raise (
168.             Failure ("return gives " ^ string_of_typ t ^ " expected " ^
169.                 string_of_typ func.typ ^ " in " ^ string_of_expr e))
170.
171.               (* A block is correct if each statement is correct and nothing
172.                  follows any Return statement.  Nested blocks are flattened. *
    )
173.             | Block sl ->
174.                 let rec check_stmt_list = function
175.                     [Return _ as s] -> [check_stmt s]
176.                   | Return _ :: _   -
    > raise (Failure "nothing may follow a return")
177.                   | Block sl :: ss  -
    > check_stmt_list (sl @ ss) (* Flatten blocks *)
178.                   | s :: ss         -> check_stmt s :: check_stmt_list ss
179.                   | []              -> []
180.                 in SBlock(check_stmt_list sl)
181.
182.           in (* body of check_function *)
183.           { styp = func.typ;
```

```
184.             sfname = func.fname;
185.             sformals = func.formals;
186.             slocals  = func.locals;
187.             sbody = match check_stmt (Block func.body) with
188.           SBlock(sl) -> sl
189.             | _ -
    > raise (Failure ("internal error: block didn't become a block?"))
190.           }
191.         in (globals, List.map check_function functions)
```

8.5 codegen.ml

```
1.  (* Code generation: translate takes a semantically checked AST and
2.     produces LLVM IR
3.  *)
4.
5.  module L = Llvm
6.  module A = Ast
7.  open Sast
8.
9.  module StringMap = Map.Make(String)
10.
11. (* translate : Sast.program -> Llvm.module *)
12. let translate (globals, functions) =
13.   let context    = L.global_context () in
14.
15.   (* Create the LLVM compilation module into which
16.      we will generate code *)
17.   let the_module = L.create_module context "PyLit" in
18.
19.   (* Get types from the context *)
20.   let i32_t      = L.i32_type    context
21.   and i8_t       = L.i8_type     context
22.   and i1_t       = L.i1_type     context
23.   and float_t    = L.double_type context
24.   and string_t   = L.pointer_type (L.i8_type context)
25.   and none_t     = L.void_type   context
26.     in
27.
28.   (* Return the LLVM type for a PyLit type *)
29.   let rec ltype_of_typ = function
30.       A.Int   -> i32_t
31.     | A.Bool  -> i1_t
32.     | A.Float -> float_t
33.     | A.None  -> none_t
34.     | A.String -> string_t
35.     | A.List(t) -> L.pointer_type (ltype_of_typ t)
36.   in
37.
38.   (* Create a map of global variables after creating each *)
39.   let global_vars : L.llvalue StringMap.t =
40.     let global_var m (t, n) =
41.       let init = match t with
42.           A.Float -> L.const_float (ltype_of_typ t) 0.0
43.         | _ -> L.const_int (ltype_of_typ t) 0
44.       in StringMap.add n (L.define_global n init the_module) m in
45.     List.fold_left global_var StringMap.empty globals in
46.
47.   let printf_t : L.lltype =
48.     L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
```

```
49.   let printf_func : L.llvalue =
50.     L.declare_function "printf" printf_t the_module in
51.
52.   let printbig_t : L.lltype =
53.     L.function_type i32_t [| i32_t |] in
54.   let printbig_func : L.llvalue =
55.     L.declare_function "printbig" printbig_t the_module in
56.
57.   let string_concat_t : L.lltype =
58.     L.function_type string_t [| string_t; string_t |] in
59.   let string_concat_f : L.llvalue =
60.     L.declare_function "string_concat" string_concat_t the_module in
61.
62.   (* let list_init_t = L.function_type lst_t [||] in
63.   let list_init_f = L.declare_function "list_init" list_init_t the_module in *)

64.
65.   (* Define each function (arguments and return type) so we can
66.      call it even before we've created its body *)
67.   let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
68.     let function_decl m fdecl =
69.       let name = fdecl.sfname
70.       and formal_types =
71.         Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
72.       in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
73.       StringMap.add name (L.define_function name ftype the_module, fdecl) m in
74.     List.fold_left function_decl StringMap.empty functions in
75.
76.   (* Fill in the body of the given function *)
77.   let build_function_body fdecl =
78.     let (the_function, _) = StringMap.find fdecl.sfname function_decls in
79.     let builder = L.builder_at_end context (L.entry_block the_function) in
80.
81.     let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
82.     and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
83.     and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder in
84.
85.     (* Construct the function's "locals": formal arguments and locally
86.        declared variables.  Allocate each on the stack, initialize their
87.        value, if appropriate, and remember their values in the "locals" map *)
88.     let local_vars =
89.       let add_formal m (t, n) p =
90.         L.set_value_name n p;
91.         let local = L.build_alloca (ltype_of_typ t) n builder in
92.         ignore (L.build_store p local builder);
93.         StringMap.add n local m
94.
95.       (* Allocate space for any locally declared variables and add the
96.        * resulting registers to our map *)
97.       and add_local m (t, n) =
98.         let local_var = L.build_alloca (ltype_of_typ t) n builder
99.         in StringMap.add n local_var m
100.             in
101.
102.             let formals = List.fold_left2 add_formal StringMap.empty fdecl.sfo
    rmals
103.                 (Array.to_list (L.params the_function)) in
104.             List.fold_left add_local formals fdecl.slocals
105.           in
106.
107.           (* Return the value for a variable or formal argument.
```

```
108.               Check local names first, then global names *)
109.           let lookup n = try StringMap.find n local_vars
110.             with Not_found -> StringMap.find n global_vars
111.           in
112.
113.           (* Construct code for an expression; return its value *)
114.           let rec expr builder ((_, e) : sexpr) = match e with
115.               SLiteral i  -> L.const_int i32_t i
116.             | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
117.             | SStringLit s -> L.build_global_stringptr s "str" builder
118.             | SFliteral l -> L.const_float_of_string float_t l
119.             | SNoexpr     -> L.const_int i32_t 0
120.             | SId s       -> L.build_load (lookup s) s builder
121.             | SAssign (s, e) -> let e' = expr builder e in
122.               ignore(L.build_store e' (lookup s) builder); e'
123.             | SBinop ((A.Float,_ ) as e1, op, e2) ->
124.               let e1' = expr builder e1
125.               and e2' = expr builder e2 in
126.               (match op with
127.                 A.Add     -> L.build_fadd
128.               | A.Sub     -> L.build_fsub
129.               | A.Mult    -> L.build_fmul
130.               | A.Div     -> L.build_fdiv
131.               | A.Mod     -> L.build_frem
132.               | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
133.               | A.Neq     -> L.build_fcmp L.Fcmp.One
134.               | A.Less    -> L.build_fcmp L.Fcmp.Olt
135.               | A.Leq     -> L.build_fcmp L.Fcmp.Ole
136.               | A.Greater -> L.build_fcmp L.Fcmp.Ogt
137.               | A.Geq     -> L.build_fcmp L.Fcmp.Oge
138.               | A.And | A.Or ->
139.                 raise (Failure "internal error: semant should have rejected a
    nd/or on float")
140.               ) e1' e2' "tmp" builder
141.             | SBinop ((A.String,_ ) as e1, op, e2) ->
142.               let e1' = expr builder e1
143.               and e2' = expr builder e2 in
144.               (match op with
145.                 A.Add     -
    > L.build_call string_concat_f [| e1'; e2' |] "string_concat" builder
146.               | _ -
    > raise (Failure ("operation " ^ (A.string_of_op op) ^ " not implemented")))
147.             | SBinop (e1, op, e2) ->
148.               let e1' = expr builder e1
149.               and e2' = expr builder e2 in
150.               (match op with
151.                 A.Add     -> L.build_add
152.               | A.Sub     -> L.build_sub
153.               | A.Mult    -> L.build_mul
154.               | A.Div     -> L.build_sdiv
155.               | A.Mod     -> L.build_srem
156.               | A.And     -> L.build_and
157.               | A.Or      -> L.build_or
158.               | A.Equal   -> L.build_icmp L.Icmp.Eq
159.               | A.Neq     -> L.build_icmp L.Icmp.Ne
160.               | A.Less    -> L.build_icmp L.Icmp.Slt
161.               | A.Leq     -> L.build_icmp L.Icmp.Sle
162.               | A.Greater -> L.build_icmp L.Icmp.Sgt
163.               | A.Geq     -> L.build_icmp L.Icmp.Sge
164.               ) e1' e2' "tmp" builder
165.             | SUnop(op, ((t, _) as e)) ->
```

```
166.                let e' = expr builder e in
167.                (match op with
168.                   A.Neg when t = A.Float -> L.build_fneg
169.                 | A.Neg                  -> L.build_neg
170.                 | A.Not                  -> L.build_not) e' "tmp" builder
171.              | SCall ("print", [e]) | SCall ("printb", [e]) ->
172.                L.build_call printf_func [| int_format_str ; (expr builder e) |]
    "printf" builder
173.              | SCall ("printbig", [e]) ->
174.                L.build_call printbig_func [| (expr builder e) |] "printbig" bui
    lder
175.              | SCall ("prints", [e]) ->
176.                L.build_call printf_func [| string_format_str ; (expr builder e)
    |] "printf" builder
177.              | SCall ("printf", [e]) ->
178.                L.build_call printf_func [| float_format_str ; (expr builder e)
    |] "printf" builder
179.              | SCall (f, args) ->
180.                let (fdef, fdecl) = StringMap.find f function_decls in
181.                let llargs = List.rev (List.map (expr builder) (List.rev args))
    in
182.                let result = (match fdecl.styp with
183.                      A.None -> ""
184.                    | _ -> f ^ "_result") in
185.                L.build_call fdef (Array.of_list llargs) result builder
186.            in
187.
188.            (* LLVM insists each basic block end with exactly one "terminator"
189.               instruction that transfers control.  This function runs "instr bu
    ilder"
190.               if the current block does not already have a terminator.  Used,
191.               e.g., to handle the "fall off the end of the function" case. *)
192.            let add_terminal builder instr =
193.              match L.block_terminator (L.insertion_block builder) with
194.                Some _ -> ()
195.              | None -> ignore (instr builder) in
196.
197.            (* Build the code for the given statement; return the builder for
198.               the statement's successor (i.e., the next instruction will be bui
    lt
199.               after the one generated by this call) *)
200.
201.            let rec stmt builder = function
202.                SBlock sl -> List.fold_left stmt builder sl
203.              | SExpr e -> ignore(expr builder e); builder
204.              | SReturn e -> ignore(match fdecl.styp with
205.                    (* Special "return nothing" instr *)
206.                      A.None -> L.build_ret_void builder
207.                    (* Build return statement *)
208.                    | _ -> L.build_ret (expr builder e) builder );
209.                  builder
210.              | SIf (predicate, then_stmt, else_stmt) ->
211.                  let bool_val = expr builder predicate in
212.                  let merge_bb = L.append_block context "merge" the_function in
213.                  let build_br_merge = L.build_br merge_bb in (* partial function
    *)
214.
215.                  let then_bb = L.append_block context "then" the_function in
216.                  add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)

217.                    build_br_merge;
```

```
218.
219.               let else_bb = L.append_block context "else" the_function in
220.               add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
221.                 build_br_merge;
222.
223.               ignore(L.build_cond_br bool_val then_bb else_bb builder);
224.               L.builder_at_end context merge_bb
225.
226.             | SWhile (predicate, body) ->
227.               let pred_bb = L.append_block context "while" the_function in
228.               ignore(L.build_br pred_bb builder);
229.
230.               let body_bb = L.append_block context "while_body" the_function i
    n
231.               add_terminal (stmt (L.builder_at_end context body_bb) body)
232.                 (L.build_br pred_bb);
233.
234.               let pred_builder = L.builder_at_end context pred_bb in
235.               let bool_val = expr pred_builder predicate in
236.
237.               let merge_bb = L.append_block context "merge" the_function in
238.               ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
239.               L.builder_at_end context merge_bb
240.
241.             (* Implement for loops as while loops *)
242.             | SFor (e1, e2, e3, body) -> stmt builder
243.                                             ( SBlock [SExpr e1 ; SWhile (e2, SB
    lock [body ; SExpr e3]) ] )
244.           in
245.
246.           (* Build the code for each statement in the function *)
247.           let builder = stmt builder (SBlock fdecl.sbody) in
248.
249.           (* Add a return if the last block falls off the end *)
250.           add_terminal builder (match fdecl.styp with
251.                 A.None -> L.build_ret_void
252.               | A.Float -> L.build_ret (L.const_float float_t 0.0)
253.               | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
254.         in
255.
256.         List.iter build_function_body functions;
257.         the_module
```

8.6 functions.c

```
1.  /*
2.   *  A function illustrating how to link C code to code generated from LLVM
3.   */
4.
5.  #include <stdio.h>
6.  #include <stdlib.h>
7.  #include <string.h>
8.
9.  char *string_concat(char *s1, char *s2) {
10.     char *new = (char *) malloc(strlen(s1) + strlen(s2) + 1);
11.     strcpy(new, s1);
12.     strcat(new, s2);
13.     return new;
```

```
14. }
15.
16. typedef struct List_element {
17.     struct List_element *next;
18.     struct List_element *prev;
19.     void *data;
20. } list_element;
21.
22. typedef struct List {
23.     int32_t length;
24.     list_element *head;
25.     list_element *tail;
26. } list;
27.
28. //init list
29. list* list_init() {
30.     list* l = (list *) malloc(sizeof(list));
31.     l->length = 0;
32.     l->head = NULL;
33.     l->tail = NULL;
34.     return l;
35. }
36.
37. /*
38.  * Font information: one byte per row, 8 rows per character
39.  * In order, space, 0-9, A-Z
40.  */
41. static const char font[] = {
42.   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
43.   0x1c, 0x3e, 0x61, 0x41, 0x43, 0x3e, 0x1c, 0x00,
44.   0x00, 0x40, 0x42, 0x7f, 0x7f, 0x40, 0x40, 0x00,
45.   0x62, 0x73, 0x79, 0x59, 0x5d, 0x4f, 0x46, 0x00,
46.   0x20, 0x61, 0x49, 0x4d, 0x4f, 0x7b, 0x31, 0x00,
47.   0x18, 0x1c, 0x16, 0x13, 0x7f, 0x7f, 0x10, 0x00,
48.   0x27, 0x67, 0x45, 0x45, 0x45, 0x7d, 0x38, 0x00,
49.   0x3c, 0x7e, 0x4b, 0x49, 0x49, 0x79, 0x30, 0x00,
50.   0x03, 0x03, 0x71, 0x79, 0x0d, 0x07, 0x03, 0x00,
51.   0x36, 0x4f, 0x4d, 0x59, 0x59, 0x76, 0x30, 0x00,
52.   0x06, 0x4f, 0x49, 0x49, 0x69, 0x3f, 0x1e, 0x00,
53.   0x7c, 0x7e, 0x13, 0x11, 0x13, 0x7e, 0x7c, 0x00,
54.   0x7f, 0x7f, 0x49, 0x49, 0x49, 0x7f, 0x36, 0x00,
55.   0x1c, 0x3e, 0x63, 0x41, 0x41, 0x63, 0x22, 0x00,
56.   0x7f, 0x7f, 0x41, 0x41, 0x63, 0x3e, 0x1c, 0x00,
57.   0x00, 0x7f, 0x7f, 0x49, 0x49, 0x49, 0x41, 0x00,
58.   0x7f, 0x7f, 0x09, 0x09, 0x09, 0x09, 0x01, 0x00,
59.   0x1c, 0x3e, 0x63, 0x41, 0x49, 0x79, 0x79, 0x00,
60.   0x7f, 0x7f, 0x08, 0x08, 0x08, 0x7f, 0x7f, 0x00,
61.   0x00, 0x41, 0x41, 0x7f, 0x7f, 0x41, 0x41, 0x00,
62.   0x20, 0x60, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
63.   0x7f, 0x7f, 0x18, 0x3c, 0x76, 0x63, 0x41, 0x00,
64.   0x00, 0x7f, 0x7f, 0x40, 0x40, 0x40, 0x40, 0x00,
65.   0x7f, 0x7f, 0x0e, 0x1c, 0x0e, 0x7f, 0x7f, 0x00,
66.   0x7f, 0x7f, 0x0e, 0x1c, 0x38, 0x7f, 0x7f, 0x00,
67.   0x3e, 0x7f, 0x41, 0x41, 0x41, 0x7f, 0x3e, 0x00,
68.   0x7f, 0x7f, 0x11, 0x11, 0x11, 0x1f, 0x0e, 0x00,
69.   0x3e, 0x7f, 0x41, 0x51, 0x71, 0x3f, 0x5e, 0x00,
70.   0x7f, 0x7f, 0x11, 0x31, 0x79, 0x6f, 0x4e, 0x00,
71.   0x26, 0x6f, 0x49, 0x49, 0x4b, 0x7a, 0x30, 0x00,
72.   0x00, 0x01, 0x01, 0x7f, 0x7f, 0x01, 0x01, 0x00,
73.   0x3f, 0x7f, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
74.   0x0f, 0x1f, 0x38, 0x70, 0x38, 0x1f, 0x0f, 0x00,
```

```
  0x1f, 0x7f, 0x38, 0x1c, 0x38, 0x7f, 0x1f, 0x00,
  0x63, 0x77, 0x3e, 0x1c, 0x3e, 0x77, 0x63, 0x00,
  0x00, 0x03, 0x0f, 0x78, 0x78, 0x0f, 0x03, 0x00,
  0x61, 0x71, 0x79, 0x5d, 0x4f, 0x47, 0x43, 0x00
};

void printbig(int c)
{
  int index = 0;
  int col, data;
  if (c >= '0' && c <= '9') index = 8 + (c - '0') * 8;
  else if (c >= 'A' && c <= 'Z') index = 88 + (c - 'A') * 8;
  do {
    data = font[index++];
    for (col = 0 ; col < 8 ; data <<= 1, col++) {
      char d = data & 0x80 ? 'X' : ' ';
      putchar(d); putchar(d);
    }
    putchar('\n');
  } while (index & 0x7);
}


#ifdef BUILD_TEST
int main()
      {
        char s[] = "HELLO WORLD09AZ";
        char *c;
        for ( c = s ; *c ; c++) printbig(*c);
      }
      #endif
```

8.7 testall.sh

```
#!/bin/sh

# Regression testing script for PyLit
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the PyLit compiler.  Usually "./pylit.native"
# Try "_build/pylit.native" if ocamlbuild was unable to create a symbolic link.

PYLIT="./pylit.native"
#PYLIT="_build/pylit.native"

# Set time limit for all operations
ulimit -t 30

```

```
globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.pyl files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    SignalError "$1 failed on $*"
    return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
    SignalError "failed: $* did not report an error"
    return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.pyl//'`
    reffile=`echo $1 | sed 's/.pyl$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

```

```
87.     echo -n "$basename..."
88.
89.     echo 1>&2
90.     echo "###### Testing $basename" 1>&2
91.
92.     generatedfiles=""
93.
94.     generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
    ${basename}.out" &&
95.     Run "$PYLIT" "$1" ">" "${basename}.ll" &&
96.     Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
97.     Run "$CC" "-o" "${basename}.exe" "${basename}.s" "functions.o" &&
98.     Run "./${basename}.exe" > "${basename}.out" &&
99.     Compare ${basename}.out ${reffile}.out ${basename}.diff
100.
101.            # Report the status and clean up the generated files
102.
103.            if [ $error -eq 0 ] ; then
104.            if [ $keep -eq 0 ] ; then
105.                rm -f $generatedfiles
106.            fi
107.            echo "OK"
108.            echo "###### SUCCESS" 1>&2
109.            else
110.            echo "###### FAILED" 1>&2
111.            globalerror=$error
112.            fi
113.        }
114.
115.        CheckFail() {
116.            error=0
117.            basename=`echo $1 | sed 's/.*\\///
118.                                    s/.pyl//'`
119.            reffile=`echo $1 | sed 's/.pyl$//'`
120.            basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
121.
122.            echo -n "$basename..."
123.
124.            echo 1>&2
125.            echo "###### Testing $basename" 1>&2
126.
127.            generatedfiles=""
128.
129.            generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&

130.            RunFail "$PYLIT" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
131.            Compare ${basename}.err ${reffile}.err ${basename}.diff
132.
133.            # Report the status and clean up the generated files
134.
135.            if [ $error -eq 0 ] ; then
136.            if [ $keep -eq 0 ] ; then
137.                rm -f $generatedfiles
138.            fi
139.            echo "OK"
140.            echo "###### SUCCESS" 1>&2
141.            else
142.            echo "###### FAILED" 1>&2
143.            globalerror=$error
144.            fi
145.        }
```

```
146.
147.        while getopts kdpsh c; do
148.            case $c in
149.            k) # Keep intermediate files
150.                keep=1
151.                ;;
152.            h) # Help
153.                Usage
154.                ;;
155.            esac
156.        done
157.
158.        shift `expr $OPTIND - 1`
159.
160.        LLIFail() {
161.          echo "Could not find the LLVM interpreter \"$LLI\"."
162.          echo "Check your LLVM installation and/or modify the LLI variable in t
    estall.sh"
163.          exit 1
164.        }
165.
166.        which "$LLI" >> $globallog || LLIFail
167.
168.        if [ ! -f functions.o ]
169.        then
170.            echo "Could not find functions.o"
171.            echo "Try \"make functions.o\""
172.            exit 1
173.        fi
174.
175.        if [ $# -ge 1 ]
176.        then
177.            files=$@
178.        else
179.            files="tests/test-*.pyl tests/fail-*.pyl"
180.        fi
181.
182.        for file in $files
183.        do
184.            case $file in
185.            *test-*)
186.                Check $file 2>> $globallog
187.                ;;
188.            *fail-*)
189.                CheckFail $file 2>> $globallog
190.                ;;
191.            *)
192.                echo "unknown file type $file"
193.                globalerror=1
194.                ;;
195.            esac
196.        done
197.
198.        exit $globalerror
```

8.8 Makefile

```
1.  # "make test" Compiles everything and runs the regression tests
2.
3.  .PHONY : test
```

```
4.  test : all testall.sh
5.      ./testall.sh
6.
7.  .PHONY : all
8.  all : pylit.native functions.o
9.
10. pylit.native :
11.     opam config exec -- \
12.     rm -f *.o
13.     ocamlbuild -use-ocamlfind -pkgs llvm.bitreader pylit.native
14.     gcc -c functions.c
15.     clang -emit-llvm -o functions.bc -c functions.c -Wno-varargs
16.
17. # "make clean" removes all generated files
18.
19. .PHONY : clean
20. clean :
21.     ocamlbuild -clean
22.     rm -rf testall.log *.diff pylit scanner.ml parser.ml parser.mli
23.     rm -rf *.cmx *.cmi *.cmo *.cmx *.ll
24.     rm -rf *.err *.out *.exe *.s
25.     rm -f *.o functions.bc
26.
27. # Testing the "printbig" example
28.
29. printbig : functions.c
30.     cc -o functions -DBUILD_TEST functions.c
31.
32. # Building the tarball
33.
34. TESTS = \
35.     add1 arith1 arith2 arith3 fib float1 float2 float3 for1 for2 func1 func3 fun
    c4 \
36.     func5 func6 func7 func8 func9 gcd gcd2 global1 global2 global3 hello hello2
    \
37.     if1 if2 if3 if4 if5 if6 local1 local2 ops1 ops2 remainder stringconcat \
38.     stringconcat2 var1 var2
39.
40. FAILS = \
41.     assign1 assign2 assign3 assign4 dead1 expr1 expr2 float1 for1 for2 for3 func
    1  \
42.     func2 func3 func4 func5 func6 func7 global1 global2 if1 if2 if3 nomain \
43.     return1 return2 while1 while2
44.
45.
46. TESTFILES = $(TESTS:%=test-%.pyl) $(TESTS:%=test-%.out) \
47.         $(FAILS:%=fail-%.pyl) $(FAILS:%=fail-%.err)
48.
49. TARFILES = ast.ml sast.ml codegen.ml Makefile _tags pylit.ml pylitparse.mly \
50.     README scanner.mll semant.ml testall.sh \
51.     functions.c arcade-font.pbm font2c \
52.     Dockerfile \
53.     $(TESTFILES:%=tests/%)
54.
55. pylit.tar.gz : $(TARFILES)
56.     cd .. && tar czf pylit/pylit.tar.gz \
57.         $(TARFILES:%=pylit/%)
```

8.9 **Test cases**

fail-assign1.err

```
1. Fatal error: exception Failure("illegal assignment int = bool in i = false")
```

fail-assign1.pyl

```
1. def int main():
2.   int i;
3.   bool b;
4.
5.   i = 42;
6.   i = 10;
7.   b = true;
8.   b = false;
9.   i = false; #Fail: assigning a bool to an integer
10. end
```

fail-assign2.err

```
1. Fatal error: exception Failure("illegal assignment bool = int in b = 48")
```

fail-assign2.pyl

```
1. def int main():
2.   int i;
3.   bool b;
4.   b = 48; #Fail: assigning an integer to a bool
5. end
```

fail-assign3.err

```
1. Fatal error: exception Failure("illegal assignment int = none in i = noneFunc()"
   )
```

fail-assign3.pyl

```
1. def none noneFunc():
2.   end
3.
4. def int main():
5.   int i;
6.   i = noneFunc(); #Fail: assigning a none to an integer
7. end
```

fail-assign4.err

```
1. Fatal error: exception Failure("illegal assignment bool = str in b = "test strin
   g"")
```

fail-assign4.pyl

```
1.  def int main():
2.    str i;
3.    bool b;
4.    b = "test string"; #Fail: assigning a string to a bool
5.  end
```

fail-dead1.err

```
1.  Fatal error: exception Failure("nothing may follow a return")
```

fail-dead1.pyl

```
1.  def int main():
2.    int i;
3.    i = 15;
4.    return i;
5.    i = 32; #Error: code after a return
6.  end
```

fail-expr1.err

```
1.  Fatal error: exception Failure("illegal binary operator bool + int in d + a")
```

fail-expr1.pyl

```
1.  int a;
2.  bool b;
3.
4.  def none foo(int c, bool d):
5.    int dd;
6.    bool e;
7.    a + c;
8.    c - a;
9.    a * 3;
10.   c / 2;
11.   d + a; #Error: bool + int
12. end
13.
14. def int main():
15.   end
```

fail-expr2.err

```
1.  Fatal error: exception Failure("illegal binary operator bool + int in b + a")
```

fail-expr2.pyl

```
int a;
bool b;

def none foo(int c, bool d):
  int d;
  bool e;
  b + a; #Error: bool + int
end

def int main():
  return 0;
end
```

fail-float1.err

```
Fatal error: exception Failure("illegal binary operator float && int in -
3.5 && 1")
```

fail-float1.pyl

```
def int main():
  -3.5 && 1;
  return 0;
end
```

fail-for1.err

```
Fatal error: exception Failure("undeclared identifier j")
```

fail-for1.pyl

```
def int main():
  int i;
  for (i = 0; j < 10 ; i = i + 1): #j undefined
    end
  return 0;
end
```

fail-for2.err

```
Fatal error: exception Failure("expected Boolean expression in i")
```

fail-for2.pyl

```
def int main():
  int i;
  for (i = 0; i ; i = i + 1): # i is an integer, not Boolean
    end
  return 0;
```

```
6.  end
```

fail-for3.err

```
1.  Fatal error: exception Failure("unrecognized function foo")
```

fail-for3.pyl

```
1.  def int main():
2.    int i;
3.    for (i = 0; i < 10; i = i + 1):
4.      foo(); #Error: no function foo
5.    end
6.    return 0;
7.  end
```

fail-func1.err

```
1.  Fatal error: exception Failure("duplicate function bar")
```

fail-func1.pyl

```
1.  def int foo():
2.    end
3.
4.  def int bar():
5.    end
6.
7.  def int baz():
8.    end
9.
10. def none bar(): #Error: duplicate function bar
11.   end
12.
13. def int main():
14.   return 0;
15. end
```

fail-func2.err

```
1.  Fatal error: exception Failure("duplicate formal a")
```

fail-func2.pyl

```
1.  def int foo(int a, bool b, int c):
2.    end
3.
4.  def none bar(int a, bool b, int a): #Error: duplicate formal a in bar
5.    end
6.
7.  def int main():
8.    return 0;
9.  end
```

fail-func3.err

```
Fatal error: exception Failure("illegal none formal b")
```

fail-func3.pyl

```
def int foo(int a, bool b, int c):
  end

def none bar(int a, none b, int c): #Error: illegal none formal b
  end

def int main():
  return 0;
end
```

fail-func4.err

```
Fatal error: exception Failure("function print may not be defined")
```

fail-func4.pyl

```
def int foo():
  end

def none bar():
  end

def int print(): #Should not be able to define print
  end

def none baz():
  end

def int main():
  return 0;
end
```

fail-func5.err

```
Fatal error: exception Failure("illegal none local b")
```

fail-func5.pyl

```
def int foo():
  end

def int bar():
  int a;
  none b; #Error: illegal none local b
  bool c;

  return 0;
```

```
10. end
11.
12. def int main():
13.   return 0;
14. end
```

fail-func6.err

```
1.  Fatal error: exception Failure("expecting 2 arguments in foo(42)")
```

fail-func6.pyl

```
1.  def none foo(int a, bool b):
2.    end
3.
4.  def int main():
5.    foo(42, true);
6.    foo(42); #Wrong number of arguments
7.  end
```

fail-func7.err

```
1.  Fatal error: exception Failure("illegal argument found int expected bool in 42")
```

fail-func7.pyl

```
1.  def none foo(int a, bool b):
2.    end
3.
4.  def int main():
5.    foo(42, true);
6.    foo(42, 42); #Fail: int, not bool
7.  end
```

fail-global1.err

```
1.  Fatal error: exception Failure("illegal none global a")
```

fail-global1.pyl

```
1.  int c;
2.  bool b;
3.  none a; #global variables should not be none
4.
5.  def int main():
6.    return 0;
7.  end
```

fail-global2.err

```
Fatal error: exception Failure("duplicate global b")
```

fail-global2.pyl

```
int b;
bool c;
int a;
int b; #Duplicate global variable

def int main():
  return 0;
end
```

fail-if1.err

```
Fatal error: exception Failure("expected Boolean expression in 42")
```

fail-if1.pyl

```
def int main():
  if (true):
    end
  if (false):
    end
  else:
    end
  if (42): #Error: non-bool predicate
    end
end
```

fail-if2.err

```
Fatal error: exception Failure("undeclared identifier foo")
```

fail-if2.pyl

```
def int main():
  if (true):
    foo; #Error: undeclared variable
  end
end
```

fail-if3.err

```
Fatal error: exception Failure("undeclared identifier bar")
```

fail-if3.pyl

```
1.  def int main():
2.    if (true):
3.      42;
4.    end
5.    else:
6.      bar; #Error: undeclared variable
7.    end
8.  end
```

fail-nomain.err

```
1.  Fatal error: exception Failure("unrecognized function main")
```

fail-nomain.pyl

```
1.  (no code needed for this test)
```

fail-return1.err

```
1.  Fatal error: exception Failure("return gives bool expected int in true")
```

fail-return1.pyl

```
1.  def int main():
2.    return true; #Should return int
3.  end
```

fail-return2.err

```
1.  Fatal error: exception Failure("return gives int expected none in 42")
```

fail-return2.pyl

```
1.  def none foo():
2.    if (true): #Should return none
3.      return 42;
4.    end
5.    else:
6.      return;
7.    end
8.  end
9.
10. def int main():
11.   return 42;
12. end
```

fail-while1.err

```
Fatal error: exception Failure("expected Boolean expression in 42")
```

fail-while1.pyl

```
def int main():
  int i;
  while (true):
    i = i + 1;
  end

  while (42): #Should be boolean
    i = i + 1;
  end
end
```

fail-while2.err

```
Fatal error: exception Failure("unrecognized function foo")
```

fail-while2.pyl

```
def int main():
  int i;
  while (true):
    i = i + 1;
  end

  while (true):
    foo(); #foo undefined
  end
end
```

test-add1.out

```
35
```

test-add1.pyl

```
def int add(int x, int y):
  return x + y;
end

def int main():
  print( add(15, 20) );
end
```

test-arith1.out

```
25
```

test-arith1.pyl

```
def int main():
  print(20 + 5);
end
```

test-arith2.out

```
11
```

test-arith2.pyl

```
def int main():
  print(1 + 2 * 3 + 4);
end
```

test-arith3.out

```
55
```

test-arith3.pyl

```
def int foo(int a):
  return a;
end

def int main():
  int a;
  a = 50;
  a = a + 5;
  print(a);
end
```

test-fib.out

```
def int foo(int a):
  return a;
end

def int main():
  int a;
  a = 50;
  a = a + 5;
  print(a);
end
```

test-fib.pyl

```
def int fib(int x):
  if (x < 2):
    return 1;
```

```
4.    end
5.    return fib(x-1) + fib(x-2);
6.  end
7.
8.  def int main():
9.    print(fib(0));
10.   print(fib(1));
11.   print(fib(2));
12.   print(fib(3));
13.   print(fib(4));
14.   print(fib(5));
15. end
```

test-float1.out

```
1.  5.12345
```

test-float1.pyl

```
1.  def int main():
2.    float a;
3.    a = 5.123456789;
4.    printf(a);
5.  end
```

test-float2.out

```
1.  3.00001
```

test-float2.pyl

```
1.  def int main():
2.    float a;
3.    float b;
4.    float c;
5.    a = 5.12345678;
6.    b = -2.12345;
7.    c = a + b;
8.    printf(c);
9.  end
```

test-float3.out

```
1.  30.1234
2.  19.8766
3.  128.086
4.  4.87952
5.  0
6.  1
7.  1
8.  0
9.  1
10. 1
11. 0
```

```
0
10.2469
0
26.2497
1
1
1
0
0
0
1
0
1
```

test-float3.pyl

```
def none testfloat(float a, float b):
  printf(a + b);
  printf(a - b);
  printf(a * b);
  printf(a / b);
  printb(a == b);
  printb(a == a);
  printb(a != b);
  printb(a != a);
  printb(a > b);
  printb(a >= b);
  printb(a < b);
  printb(a <= b);
end

def int main():
  float c;
  float d;

  c = 25.0;
  d = 5.12345;

  testfloat(c, d);
  testfloat(d, d);

end
```

test-for1.out

```
0
1
2
3
4
12
```

test-for1.pyl

```
def int main():
  int i;
  for (i = 0 ; i < 5 ; i = i + 1):
```

```
4.      print(i);
5.    end
6.    print(12);
7.  end
```

test-for2.out

```
1.  0
2.  1
3.  2
4.  3
5.  4
6.  20
```

test-for2.pyl

```
1.  def int main():
2.    int i;
3.    i = 0;
4.    for ( ; i < 5; ):
5.      print(i);
6.      i = i + 1;
7.    end
8.    print(20);
9.  end
```

test-for3.out

```
1.  0
2.  1
3.  2
4.  3
5.  4
6.  55
```

test-for3.pyl

```
1.  def int main():
2.    int i;
3.      for (i = 0 ; i < 5 ; i = i + 1):
4.        print(i);
5.      end
6.    print(55);
7.    return 0;
8.  end
```

test-func1.out

```
1.  9
```

test-func1.pyl

```
1.  def int add(int a, int b):
```

```
2.    return a + b;
3.  end
4.
5.  def int main():
6.    int a;
7.    a = add(7, 2);
8.    print(a);
9.  end
```

test-func3.out

```
1.  15
2.  17
3.  192
4.  8
```

test-func3.pyl

```
1.  def none printem(int a, int b, int c, int d):
2.    print(a);
3.    print(b);
4.    print(c);
5.    print(d);
6.  end
7.
8.  def int main():
9.    printem(15,17,192,8);
10. end
```

test-func4.out

```
1.  62
```

test-func4.pyl

```
1.  def int add(int a, int b):
2.    int c;
3.    c = a + b;
4.    return c;
5.  end
6.
7.  def int main():
8.    int d;
9.    d = add(52, 10);
10.   print(d);
11. end
```

test-func5.out

```
1.  (output should be blank)
```

test-func5.pyl

```
def int foo(int a):
  return a;
end

def int main():
end
```

test-func6.out

```
104
```

test-func6.pyl

```
def none foo():
  end

def int bar(int a, bool b, int c):
 return a + c;
end

def int main():
  print(bar(4, false, 100));
end
```

test-func7.out

```
60
```

test-func7.pyl

```
int a;

def none foo(int c):
  a = c + 35;
end

def int main():
  foo(25);
  print(a);
end
```

test-func8.out

```
43
```

test-func8.pyl

```
def none foo(int a):
  print(a + 3);
```

```
3.  end
4.
5.  def int main():
6.    foo(40);
7.  end
```

test-func9.out

```
1.  43
```

test-func9.pyl

```
1.  def none foo(int a):
2.    print(a + 3);
3.    return;
4.  end
5.
6.  def int main():
7.    foo(40);
8.  end
```

test-gcd.out

```
1.  2
2.  3
3.  11
```

test-gcd.pyl

```
1.  def int gcd(int a, int b):
2.    while (a != b):
3.      if (a > b) a = a - b;
4.      else b = b - a;
5.    end
6.    return a;
7.  end
8.
9.  def int main():
10.   print(gcd(2,14));
11.   print(gcd(3,15));
12.   print(gcd(99,121));
13.   return 0;
14. end
```

test-gcd2.out

```
1.  7
2.  4
3.  11
```

test-gcd2.pyl

```
1.  def int gcd(int a, int b):
```

```
2.    while (a != b):
3.      if (a > b):
4.        a = a - b;
5.      end
6.      else:
7.        b = b - a;
8.      end
9.    end
10.   return a;
11. end
12.
13. def int main():
14.   print(gcd(14,21));
15.   print(gcd(8,36));
16.   print(gcd(99,121));
17. end
```

test-global1.out

```
1.  30
2.  17
3.  31
4.  18
```

test-global1.pyl

```
1.  int a;
2.  int b;
3.
4.  def none printa():
5.    print(a);
6.  end
7.
8.  def none printbb():
9.    print(b);
10. end
11.
12. def none incab():
13.   a = a + 1;
14.   b = b + 1;
15. end
16.
17. def int main():
18.   a = 30;
19.   b = 17;
20.   printa();
21.   printbb();
22.   incab();
23.   printa();
24.   printbb();
25. end
```

test-global2.out

```
1.  56
```

test-global2.pyl

```
1.  bool i;
2.
3.  def int main():
4.    int i; # Should hide the global i
5.    i = 28;
6.    print(i + i);
7.  end
```

test-global3.out

```
1.  24
```

test-global3.pyl

```
1.  int i;
2.  bool b;
3.  int j;
4.
5.  def int main():
6.    i = 14;
7.    j = 10;
8.    print(i + j);
9.  end
```

test-hello.out

```
1.  Hello World!
```

test-hello.pyl

```
1.  def none main():
2.    prints("Hello World!");
3.  end
```

test-hello2.out

```
1.  hello
```

test-hello2.pyl

```
1.  def str helloworld(str a):
2.    return a;
3.  end
4.
5.  def str main():
6.    str b;
7.    b = helloworld("hello");
8.    prints(b);
9.    return b;
```

```
10. end
```

test-if1.out

```
1.  19
2.  17
```

test-if1.pyl

```
1.  def int main():
2.    if (true):
3.      print(19);
4.    end
5.    print(17);
6.  end
```

test-if2.out

```
1.  19
2.  17
```

test-if2.pyl

```
1.  def int main():
2.    if (true):
3.      print(19);
4.    end
5.    else:
6.      print(8);
7.    end
8.    print(17);
9.  end
```

test-if3.out

```
1.  17
```

test-if3.pyl

```
1.  def int main():
2.    if (false):
3.      print(2);
4.    end
5.    print(17);
6.  end
```

test-if4.out

```
1.  8
2.  17
```

test-if4.pyl

```
def int main():
  if (false):
    print(19);
  end
  else:
    print(8);
  end
  print(17);
end
```

test-if5.out

```
19
17
```

test-if5.pyl

```
def int cond(bool b):
  int x;
  if (b)
    x = 19;
  else
    x = 17;
  return x;
end

def int main():
 print(cond(true));
 print(cond(false));
end
```

test-if6.out

```
19
10
```

test-if6.pyl

```
def int cond(bool b):
  int x;
  x = 10;
  if (b)
    if (x == 10)
      x = 19;
  else
    x = 17;
  return x;
end

def int main():
 print(cond(true));
 print(cond(false));
```

```
15. end
```

test-local1.out

```
1.  18
```

test-local1.pyl

```
1.  def none foo(bool i):
2.    int i; #Should hide the formal i
3.
4.    i = 9;
5.    print(i + i);
6.  end
7.
8.  def int main():
9.    foo(true);
10. end
```

test-local2.out

```
1.  47
```

test-local2.pyl

```
1.  def int foo(int a, bool b):
2.    int c;
3.    bool d;
4.
5.    c = a;
6.
7.    return c + 10;
8.  end
9.
10. def int main():
11.  print(foo(37, false));
12. end
```

test-ops1.out

```
1.  3
2.  -1
3.  2
4.  50
5.  99
6.  0
7.  1
8.  99
9.  1
10. 0
11. 99
12. 1
13. 0
14. 99
```

```
1
1
0
99
0
1
99
0
1
1
```

test-ops1.pyl

```
def int main():
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  printb(1 == 2);
  printb(1 == 1);
  print(99);
  printb(1 != 2);
  printb(1 != 1);
  print(99);
  printb(1 < 2);
  printb(2 < 1);
  print(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
  print(99);
  printb(1 > 2);
  printb(2 > 1);
  print(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
  return 0;
end
```

test-ops2.out

```
1
0
1
0
0
0
1
1
1
0
1
0
-10
19
```

test-ops2.pyl

```
def int main():
  printb(true);
  printb(false);
  printb(true && true);
  printb(true && false);
  printb(false && true);
  printb(false && false);
  printb(true || true);
  printb(true || false);
  printb(false || true);
  printb(false || false);
  printb(!false);
  printb(!true);
  print(-10);
  print(--19);
end
```

test-remainder.out

```
2
```

test-remainder.pyl

```
def int remainder(int a, int b):
  return a % b;
end

def int main():
  int a;
  a = remainder(38, 3);
  print(a);
end
```

test-stringconcat.out

```
hello world
```

test-stringconcat.pyl

```
def str helloworld(str a, str b):
  return a + b;
end

def str main():
  str c;
  c = helloworld("hello", " world");
  prints(c);
  return c;
end
```

test-stringconcat2.out

```
1. hello world! hello!
```

test-stringconcat2.pyl

```
1. def str main():
2.   str a;
3.   str b;
4.   a = "hello!";
5.   b = "hello" + " world! " + a;
6.   prints(b);
7.   return b;
8. end
```

test-var1.out

```
1. 19
```

test-var1.pyl

```
1. def int main():
2.   int a;
3.   a = 19;
4.   print(a);
5. end
```

test-var2.out

```
1. 92
```

test-var2.pyl

```
1. int a;
2.
3. def none foo(int c):
4.   a = c + 19;
5. end
6.
7. def int main():
8.   foo(73);
9.   print(a);
10. end
```

test-while1.out

```
1. 5
2. 4
3. 3
4. 2
5. 1
```

```
6.  19
```

test-while1.pyl

```
1.  def int main():
2.    int i;
3.    i = 5;
4.    while (i > 0):
5.      print(i);
6.      i = i - 1;
7.    end
8.    print(19);
9.  end
```

test-while2.out

```
1.  14
```

test-while2.pyl

```
1.  def int foo(int a):
2.    int j;
3.    j = 0;
4.    while (a > 0):
5.      j = j + 2;
6.      a = a - 1;
7.    end
8.    return j;
9.  end
10.
11. def int main():
12.   print(foo(7));
13. end
```

test-while3.out

```
1.  5
2.  4
3.  3
4.  2
5.  1
6.  19
```

test-while3.pyl

```
1.  def int main():
2.    int i;
3.    i = 5;
4.      while (i > 0):
5.        print(i);
6.        i = i - 1;
7.      end
8.    print(19);
9.    return 0;
```

```
10. end
```