Programming Languages & Translators

Spring 2021

The ComPyled Programming Language

Final Report

Cameron Miller (cm3959)
Daniel Hanoch (dh2964)
Gabriel Clinger (gc2821)
George DiNicola (gd2581)

1 Introduction to ComPyled

ComPyled is a language inspired by Python and C. It's syntax emulates Python in most ways, without requiring indentation to indicate blocks. ComPyled retains the syntax that makes Python an easy and fun language, but borrows convenient aspects from C's syntax. Like C, semicolons are used to indicate statement endings, as well as curly braces to indicate blocks instead of solely indentation.

1.1 Convenience

Compyled is a general-purpose language that allows the user to define and initialize functions, expressions, and statements, as well as local and global variables. The programming language also provides the user with a convenient way to declare and call objects anywhere they would like in the code; that is, there is no need to declare a function and its contents before calling it later in the program. This implementation carries out one of the most appealing functionalities in Python and gives the user the freedom of definition placement in their code.

2 ComPyled Tutorial

2.1 Environment Setup

First, clone the ComPyled GitHub repository via your terminal with the following command:

```
> git clone https://github.com/cammiller1/plt-parser.git
```

Next, ensure that you have installed Docker (code development and execution take place in the Docker container). Refer to the Dockerfile in the ComPyled repository for more details.

2.2 Build & Run ComPyled

Once you are in the correct repository, run the following commands the run the Docker container and initialize the Compyled programming environment:

> docker run --rm -it -v `pwd`:/home/plt-parser -w=/home/plt-parser columbiasedwards/plt

> make

** “make clean” after each test run to clear all executables, the build directory, and other generated files.

Using a shell script called testall.sh, the Bash shell program will execute all test cases. To make your own test create a program called <program name>.cp (abbreviation for ComPyled), and another <program name>.out as the expected output for your test.

Run: > make

The shell script will then output an “OK” or “FAILED” in the terminal:

“OK” → output matches your .out file.

“FAILED” → An error message will appear.

2.3 Let’s Run Your First Program

```
def int gcd(int a, int b){
        while (a != b){
                if (a < b) {
                        b = b - a;
                }
                else{
                        a = a - b;
                }
        }
        return a;
}

def int main()
{
  int a;
  a = gcd(12, 18);
  print(a);
  return 0;
}
main();
```

Save the code gcd.cp in the "tests" directory as well as gcd.out (should contain 6) and run "make" while inside of the Docker container.

2.4 Debugging Options

For debugging purposes, we use semant.ml and the testall.sh files to output an appropriate error message for failed tests cases. We tailored each semantic failure to output a certain message which directs the user what went wrong. After running all testall.sh the user will receive a testall.log file which details fatal errors and the reasons for them. For example:

```
###### Testing test-arith-int-exponent
./compyled.native tests/test-arith-int-exponent.cp > test-arith-int-exponent.ll
llc -relocation-model=pic test-arith-int-exponent.ll > test-arith-int-exponent.s
cc -o test-arith-int-exponent.exe test-arith-int-exponent.s
./test-arith-int-exponent.exe
diff -b test-arith-int-exponent.out tests/test-arith-int-exponent.out >
test-arith-int-exponent.diff
FAILED test-arith-int-exponent.out differs from
tests/test-arith-int-exponent.out
###### FAILED
```

Here, we test "test-arith-int-exponent" and get a "differs from" error that suggests that the output does not match the .out file we created to match it with.

3 Lexical conventions

3.1.1 Comments

Block comments can be single-line or multi-line, as long as they are enclosed by two ## symbols. For example, ## *this is a comment* ##. Compyled does not have line comments.

3.1.2 Identifiers

Identifiers follow the same conventions as Python. Valid identifiers are made of letters in lowercase (a to z) or uppercase (A to Z), digits (0 to 9), and/or an underscore _. Note that the first character must be a lowercase letter, not a digit or an underscore. For example, *varName_1* is a valid identifier. When a function is called, the function name will always be followed by parentheses containing a list of parameters, similar to Python, C, Java, etc. For example:

```
def int max(int a, int b) { … }
int max_value = max(1, 5);
print(max_value);
```

We restate this more formally below using the following regular expression.

```
[ 'a'-'z', 'A'-'Z', '0'-'9',  '_' ]*
```

3.1.3 Type Specifiers

ComPyled supports the following types: *int*, *boolean*, *float*, *string*, and *array*. There are several operators affiliated with each type. The following symbols are reserved as operators. Note that some operators are overloaded. For further explanation of how these operators are used, refer to Section 3.2.2 Unary Expressions and 3.2.3 Binary Expressions.

Integers are recognized by the regexp:	['0'-'9']+
Floats are recognized by:	['0'-'9']* '.' ['0'-'9']+ | ['0'-'9']+ '.' ['0'-'9']*

Strings are recognized by: letter = ['a'-'z' 'A'-'Z'] and string_literal = ('"'[' '-'~']*'"')

3.1.4 Operators

Note: In the usage column below, 'identifier' refers to a variable name, 'value' refers to a value of type 'T', and '⊕' *is a placeholder for an operator. '→' denotes the return type of the operation, if any.*

Usage	Operators	Description	Example

identifier ⊕ value	=	Assignment for all types	*int x;*
			string a;
			x = 6;
			a = "hello world";

int ⊕ int *→ int*	==, !=, <, <=, >, >=, *, /, +, -, %	A regular integer type int	*print(1 < 4); ##true##* *6 % 4 → evaluates to 2*

value1 ⊕ value2
→ boolean
	==, !=,
	and, or
		Evaluates to True or False	*boolean rainy = true;*
			boolean wet = false;
			boolean both = rainy and wet; ##false##

float ⊕ float *→ float*	==, !=, <, <=, >, >=, *, **, /, +, -	A double-precision float type	*0.3 + (1/5); ##0.5##*

String ⊕ String
→ String
	&	Regular string type. Characters can be represented by strings of	*String a = "Colu";*
			String b = "mbia";
			String c = a & b; ##"Columbia"##

		length 1.	

3.1.5 Keywords

ComPyled makes use of several keywords from Python. There is a keyword corresponding to each control structure, namely *for*, *while*, *if*, and *else*. Additionally *def* is reserved for function definitions, *return* is reserved for return statements. *True* and *False* are reserved to represent boolean values, and *None* denotes a null value or no value at all. *print* and *len* are the names of two built-in functions available to the user of the language. Below is a list of all the keywords that are recognized by ComPyled's scanner:

if	*else*			
while	*for*			
def	*return*			
int	*float*	*bool*	*string*	*void*
and	*or*			
True	*False*			

3.2 Expressions

3.2.1. Identifiers

An identifier is a primary expression whose type is specified by its declaration:
int x means that x is of type integer.

The grammar for expressions is as follows:

```
expr:
    ILITERAL           { Liti($1) }
  | FLITERAL           { Litf($1) }
  | BLITERAL           { Litb($1) }
  | SLITERAL           { Lits($1) }
  | ID                 { Id($1) }
  | ID LBRACKET expr RBRACKET { ArrayIndexAccess($1, $3) }
  | LBRACKET typ expr RBRACKET { LitArray($2, $3) }  /* array
decl for type and size */
  | expr PLUS   expr   { Binop($1, Add, $3) }
  | expr MINUS  expr   { Binop($1, Sub, $3) }
  | expr TIMES  expr   { Binop($1, Mul, $3) }
  | expr DIVIDE expr   { Binop($1, Div, $3) }
  | expr LT expr       { Binop($1, Lt, $3) }
  | expr GT expr       { Binop($1, Gt, $3) }
  | expr MOD expr      { Binop($1, Mod, $3) }
  | expr LTE expr      { Binop($1, Lte, $3) }
  | expr GTE expr      { Binop($1, Gte, $3) }
  | expr EQ expr       { Binop($1, Eq, $3) }
  | expr NE expr       { Binop($1, Ne, $3) }
  | expr AND expr      { Binop($1, And, $3) }
  | expr OR expr       { Binop($1, Or, $3) }
  | ID ASSIGN expr     { Assign($1, $3) }
  | ID LBRACKET expr RBRACKET ASSIGN expr { ArrayIndexAssign($1,
$3, $6) }  /* array index assign */
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }  /* function
call */
  | LPAREN expr RPAREN { $2 }
  | NOT expr           { Uniop(Not, $2) }
```

3.2.2 Unary Operators

Unary operators are recognized by the form *Uniop(OP, $2)* where *OP* represents the operation name.

The table below shows all of the unary operators that ComPyled features.

Expression	Grammar	Usage
not	NOT expr	The logical negation operator “not” changes the value of a binary expression to its opposite. If the value of an expression is True, the result of the ! expression is False. This operator is applicable only to Boolean expressions.

3.2.3 Binary Operators

Unary operators are recognized by the form *Binop($1, OP, $3)* where *OP* represents the operation name.

The table below shows all binary expressions that ComPyled features.

Expression	Grammar	Usage
= assignment operator	VARIABLE ASSIGN expr	The assignment expression, which groups right to left, takes two expressions and the value of the right expression is stored in the left expression. The type of the expression being stored must be the same as the type declaration of the expression on the left.
* multiplication	expr TIMES expr	The binary * operator multiplies the expressions. This operator is applicable only

		on numeric expressions: int, float
/ division	expr DIVIDE expr	The binary / operator divides the first expression by the right expression. This operator is applicable only on numeric expressions: int, float.
% modulo	expr MOD expr	The binary % operator gives the remainder of the division of the expressions. This operator is applicable only on numeric expressions: int, float
- subtraction	expr MINUS expr	The binary - operator subtracts the right expression from the left. This operator is applicable only on numeric expressions: int, float
+ addition	expr PLUS expr	The binary + operator adds the expressions. If the expressions are a numeric type the result is an addition. If the expressions are strings the result is a concatenation of the expressions.

3.2.4 Comparison Operators

The table below shows all comparison operators that ComPyled features.

Expression	Grammar	Usage

< less than	expr LT expr	The binary < operator returns True if the expression on the left is smaller than the expression on the right. This operator is applicable only on numeric expressions.
> greater than	expr GT expr	The binary > operator returns True if the expression on the left is larger than the expression on the right. This operator is applicable only on numeric expressions.
<= less than or equal	expr LTE expr	The binary < operator returns True if the expression on the left is smaller than or equal to the expression on the right. This operator is applicable only on numeric expressions.
>= great than or equal	expr GTE expr	The binary < operator returns True if the expression on the left is larger than or equal to the expression on the right. This operator is applicable only on numeric expressions.
== equals	expr EQ expr	The binary == operator returns True if the two expressions are equal
!= not equals	expr NE expr	The binary != operator returns True if the two expressions are not equal.
and	expr AND expr	The binary and operator returns True if both expressions evaluate to True.
or	expr OR expr	The binary or operator returns true if at least one of the expressions evaluate to True.

3.2.5 Precedence

Below is the order of precedence. The higher the operation name, the greater the precedence. Operations in the same line have equal precedence.

Assignment (=)
And (and), Or (or)
Equal (==), Not Equal (!=)
Less Than (<), Greater Than (>), Less Than Or Equal (<=), Greater Than Or Equal (>=)
Multiplication (*), Division (/), Modulo (%)
Unary Plus (++), Unary Minus (--)

3.3 Statements

3.3.1 Expression Statement

Expression statements have the form: *expression;* Expression statements are either an assignment, a unary operation, a binary operation, a function call, or a combination of these operations.

The grammar for statements is as follows:

```
stmt:
    expr SEMC { Expr $1 }
  | RETURN expr_opt SEMC { 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 SEMC expr SEMC expr RPAREN stmt {
For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
```

3.3.2 Conditional Statement

The three forms of the conditional statement are:

if (expression) {
    statement;
}

If (expression) {
    statement;
} **else** {
    statement;
}

if (expression) {
    statement;
} **else if** (expression) {
    statement;
...
else {
    statement;
}

If in any of the above cases the expression is evaluated to true, the first substatement is executed. In the second case and third case, the last substatement is executed if the previous conditions do not evaluate to true. The third case chains if conditions when there are more than two expressions to evaluate, and executes the associated substatement under that condition.

3.3.3 While Statement

The **while** statement has the form:

```
while (expression) {
        statement;
}
```

The substatement is executed continuously while the value of the expression evaluates to true. The evaluation takes place prior to the execution of each substatement inside of the loop.

3.3.4 For Statement

The **for** statement has the form:

```
for (<assignment> ; <condition> ; <increment/decrement>) {
        statement;
}
```

The <assignment> represents an initialized data type and on each iteration of the loop the <assignment> variable will be assigned to the incremented/decremented value on that specific iteration. On each iteration, the substatement is executed.

3.3.5 Return Statement

A function is terminated and returned to its caller by the **return** statement, which has one of the following forms:

---

return;
return (*expression*);
In the first case returns the None value. The second case returns the value of the expression to the function caller.

3.4 Scope

3.4.1 Lexical scope

Variables created inside a curly bracket's scope exclusively belong to the scope. The user cannot access the variable outside of the scope unless creating a new variable of the same name and/or a different value. Identifiers are permitted to be used only within their region, enclosed by curly brackets. Attempts to use an identifier outside of its region will result in an error of "undefined variable."

- Local: a local variable can be used inside the scope it was initialized
- Global: a global variable can be used anywhere within the scope in which it was initialized. It can also be used inside "indented" scopes.
  Example:

```
int x = 7;
int y = 9;
          {
                    int z = x + y;
          }
print(z)            ## will raise an error - z is undefined ##
```

3.4.2 Function declarations

def *returntype function-name*(*datatype parameter-1, datatype parameter-2, ..., datatype parameter-n) {*

statement;

}

3.4.3 Mutability

Primitives are immutable - the only way to change the value of a primitive is by creating a new object of the same name.

3.4.4 String Concatenation

For the string concatenation we overloaded the plus (+) operator to work both on strings and integers. We used a C code to establish the concatenation logic.

string word = “race”;

string other_word = “car”;

string combination = word + other_word; → evaluates to “race car”

4 Project plan

4.1 Planning and Specification

Our team met between once and twice per week in order to make constant progress with Compyled. We checked-in with our TA, Xijiao, every Saturday at 7:30 pm for a quick run-through, questions, and suggestions. We also added her as a contributor in Github so she could review what we had done so far. Initially, we had a different idea in mind and wanted to implement a relational language that focuses on relational, predicate, and propositional logics. We soon understood that as much as this idea was ambitious, it would be much trickier to carry out than we originally thought. Next, we decided to develop a statically-typed language that brings the two worlds of Python and C together. We aspired to remove the indentation constraint Python forces on us, but at the same time enjoy it's easy-to-read syntactic notation, and architectural freedom. In addition, our language allows for a new programmer to learn the semantics of C without difficult syntax, the headache of managing memory, using pointers, and the overhead of manipulating strings. C language was the perfect fit for this integration since it uses the explicit scoping design (curly brackets) and defines return types unambiguously.

4.2 Development and Testing

We tried to follow the conventional way of bottom-up development. We first develop scanner.mll and parser.mly, then the ast.ml. For our first big deadline, hello-world, we made basic structural additions to semant.ml and codegen.ml. Later, we built each layer on top of the hello-world template, one thing at a time. After having a complete compiler that could accomplish basic operations, we implemented one feature at a time.
For testing, we decided to divide the work between all group members then pushed all changes to different branches onto our main Github page. The entire test suite consists of failing test cases as well as passing test cases, all of which are assessed through testall.sh.

4.3 Software Development Tools

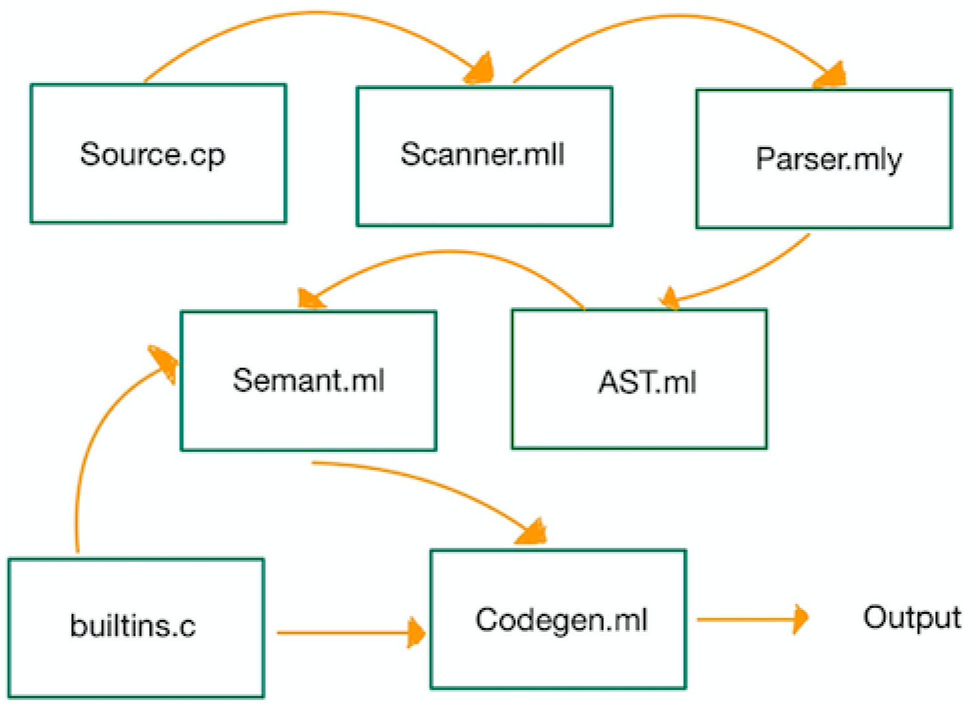
We used the following programming environments and development tools to create ComPyled:

- **Libraries and Languages:** Ocaml: Ocamlyacc & Ocammllex. LLVM.

- **Software:** Version control tool (git), Sublime Text, Visual Studio
- **Environment:** Docker container environment
- **Source Code Management:** Github

5 Language Evolution

5.1 Compiler Pipeline



5.2 Scanner

File: scanner.mll

The lexer takes a source code and translates the ASCII characters to their appropriate tokens. The scanner supports various tokens such as literals, identifiers, and keywords. It additionally tokenizes comments (## …. ##) that are being removed/rejected. All defined tokens will be used later by the parser, all undefined tokens will cause syntax errors for undefined characters.

5.3 Parser

File: parser.mly

The parser takes the legal tokens from the scanner and converts them into an Abstract Syntax Tree (AST). The conversion occurs based on the rules defined by our context-free grammar. The top level of the grammar consists of accepting statements (statements that occur outside of a function), variable declarations, and function declarations at the top level of our program.

5.4 Semantic Checking

File: semant.ml

The semantic checker converts the Abstract Syntax Tree into a Semantically-checked Abstract Syntax Tree (SAST). In case of typing or scoping errors, messages will be printed to indicate the type of errors obtained.

5.5 Code Generation

File: codegen.ml

The code generation file takes a semantically-check abstract syntax tree and converts it into its intermediate representation using LLVM. The Low Level Virtual Machine then optimizes the code and turns it into a fast-to-read assembly language.

6 Test Plan

The test plan for ComPyled was first implementing a particular scope of code or functionality and testing it against various pass/fail instances located in the test/ directory. Together with these pass/fail instances, we also tested our program against edge cases.

Before running all files together and checking whether all of our test cases are getting the “OK” or “FAILED” output, we compiled each file separately to make sure they are stable.

Working as a group, we tried to limit merge conflicts into main as much as possible, so for every new functionality we created a new branch which we worked in.

6.1 C Functions

All builtin c functions we defined are located in builtins.c. We put all logic for our length and concatenation method in builtins.c and call them using makefiles and integrating with codegen.

7 Appendix

7.1 Makefile

```
# "make test" Compiles everything and runs the regression tests

.PHONY : test
test : all testall.sh
	./testall.sh

# "make all" builds the executable

.PHONY : all
all : compyled.native builtins.o

# "make compyled.native" compiles the compiler
#
# The _tags file controls the operation of ocamlbuild, e.g., by including
# packages, enabling warnings
#
# See https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc

compyled.native :
	opam config exec -- \
	rm -f *.o
	ocamlbuild -use-ocamlfind compyled.native
	gcc -c builtins.c

# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf ocamlllvm
	rm -rf _build
	rm -rf testall.log *.diff *.ll
```

7.2 builtins.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>


/*
 * String concatenation operator
 */

char *string_concat(const char *s1, const char *s2)
{
	char *result = malloc(strlen(s1) + strlen(s2) + 1);
    strcpy(result, s1);
    strcat(result, s2);
    return result;
}
```

```
/*
 * Return the length of a given string
 */

int len(const char *s) {
    return strlen(s);
}
```

7.3 Dockerfile

```
# Based on 20.04 LTS
FROM ubuntu:focal

RUN apt-get -yq update && \
    apt-get -y upgrade && \
    apt-get -yq --no-install-suggests --no-install-recommends install \
    ocaml \
    menhir \
    llvm-10.0 \
    llvm-10.0-dev \
    m4 \
    git \
    aspcud \
    ca-certificates \
    python2.7 \
    pkg-config \
    cmake \
    opam \
    gcc \
    clang

RUN ln -s /usr/bin/lli-10.0 /usr/bin/lli
RUN ln -s /usr/bin/llc-10.0 /usr/bin/llc

RUN opam init
RUN opam install \
    llvm.10.0.0 \
    ocamlfind

WORKDIR /root

ENTRYPOINT ["opam", "config", "exec", "--"]

CMD ["bash"]
```

7.4 testall.sh

```
#!/bin/sh

# Regression testing script for ComPyled
# 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 ComPyled compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
COMPYLED="./compyled.native"
#MICROC="_build/microc.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] [.cp 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/.cp//'`
    reffile=`echo $1 | sed 's/.cp$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
${basename}.out" &&
    Run "$COMPYLED" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "builtins.o"&&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
           if [ $keep -eq 0 ] ; then
               rm -f $generatedfiles
           fi
           echo "OK"
           echo "###### SUCCESS" 1>&2
    else
```

```
        echo "###### FAILED" 1>&2
        globalerror=$error
    fi
}

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

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$COMPYLED" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
        fi
        echo "OK"
        echo "###### SUCCESS" 1>&2
    else
        echo "###### FAILED" 1>&2
        globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
        k) # Keep intermediate files
            keep=1
            ;;
        h) # Help
           Usage
           ;;
   esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail


if [ $# -ge 1 ]
then
   files=$@
else
   files="tests/test-*.cp tests/fail-*.cp"
fi

for file in $files
do
   case $file in
        *test-*)
           Check $file 2>> $globallog
           ;;
        *fail-*)
           CheckFail $file 2>> $globallog
           ;;
        *)
           echo "unknown file type $file"
           globalerror=1
           ;;
   esac
done

exit $globalerror
```

7.5 scanner.mll

```
{ open Parser
  let remove_quotes str =
  match String.length str with
  | 0 | 1 | 2 -> ""
  | len -> String.sub str 1 (len - 2)

}

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let flt = digit*'.'digit+
let string_literal = ('"'[' '-'~']*'"')

rule tokenize = parse
 [' ' '\t' '\r' '\n'] { tokenize lexbuf }
| "##"     { comment lexbuf }
| "True"   { BLITERAL (true) }
| "False"  { BLITERAL (false) }
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "void"   { VOID }
| "string" { STRING }
| "array"  { ARRAY }
| ","      { COMMA }
| "=="     { EQ }
| "=<"     { LTE }
| ">="     { GTE }
| "not"    { NOT }
| "!="     { NE }
| "and"    { AND }
| "or"     { OR }
| "in"     { IN }
| '%'      { MOD }
| '>'      { GT }
| '<'      { LT }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'     { TIMES }
| '/'     { DIVIDE }
| ';'     { SEMC }
| '='     { ASSIGN }
| '('     { LPAREN }
| ')'     { RPAREN }
| '{'     { LBRACE }
| '}'     { RBRACE }
| '['         { LBRACKET }
| ']'         { RBRACKET }
| "if"    { IF }
| "else"   { ELSE }
| "while"  { WHILE }
| "for"    { FOR }
| "def"    { DEF }
| "return" { RETURN }
| string_literal as lxm { SLITERAL(remove_quotes lxm) }
| digit+ as lxm { ILITERAL(int_of_string lxm) }
| flt as lxm { FLITERAL(lxm) }
| letter['a'-'z' 'A'-'Z' '0'-'9' '_']*  as lxm { ID(lxm) }
| eof     { EOF }
| _ as char { raise (Failure("illegal character" ^ Char.escaped char)) }

and comment = parse
  "##" { tokenize lexbuf }
| _    { comment lexbuf }
```

7.6 parser.mly

```
%{
open Ast

let fst (a,_,_) = a;;
let snd (_,b,_) = b;;
let trd (_,_,c) = c;;

%}
```

```
/* Declarations: tokens, precendence, etc. */

/* types */
%token INT FLOAT BOOL STRING VOID ARRAY
/* operators */
%token ASSIGN PLUS MINUS TIMES DIVIDE
%token SEMC
%token LBRACKET RBRACKET
/* comparators */
%token LT GT LTE GTE EQ NE AND OR MOD NOT IN

%token IF ELSE WHILE FOR DEF RETURN
%token LPAREN RPAREN RBRACE LBRACE COMMA

%token <int> ILITERAL
%token <string> FLITERAL
%token <string> ID
%token <string> SLITERAL
%token <bool> BLITERAL
%token EOF

/* lowest to highest precedence */
%nonassoc NOELSE
%nonassoc ELSE
%right UMINUS
%right ASSIGN
%left AND OR
%left EQ NE
%left LT GT LTE GTE
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT

%start program  /* the entry point */
%type <Ast.program> program


%%
/* Rules: context-free rules */
```

```
program:
  highest EOF { $1 }


highest:
   /* nothing */ { ([], [], [])            }
 | highest vdecl { (($2 :: fst $1), snd $1, trd $1) }
 | highest fdecl { (fst $1, ($2 :: snd $1), trd $1) }
 | highest stmt { (fst $1, snd $1, (trd $1 @ [$2]))  }


/* statement-relevant parsing */
stmt:
    expr SEMC { Expr $1 }
  | RETURN expr_opt SEMC { 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 SEMC expr SEMC expr RPAREN stmt { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
/* ======================================== */

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


vdecl:
     typ ID vdecl_assign SEMC { ($1, $2, $3) } /* declaration with or w.o assignment */

vdecl_assign:
                 { Noexpr }
  | ASSIGN expr { $2 }
  | expr  { $1 }


vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }
```

```
fdecl:
  DEF typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
  { { typ = $2;
        fname = $3;
        formals = List.rev $5;
        locals = List.rev $8;
        body = List.rev $9 } }

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

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


/* type-relevant parsing */
typ:
    INT            { Int }
  | FLOAT          { Float }
  | BOOL           { Boolean }
  | STRING         { String }
  | VOID           { Void }
  | ARRAY          { Array }
/* ======================================= */



/* expression-relevant parsing */
expr_opt:
    /* nothing */ { Noexpr }
  | expr         { $1 }

expr:
    ILITERAL           { Liti($1) }
  | FLITERAL           { Litf($1) }
  | BLITERAL           { Litb($1) }
  | SLITERAL           { Lits($1) }
  | ID                 { Id($1) }
  | ID LBRACKET expr RBRACKET { ArrayIndexAccess($1, $3) }
  | LBRACKET typ expr RBRACKET { LitArray($2, $3) }  /* array decl for type and size */
  | expr PLUS   expr   { Binop($1, Add, $3) }
  | expr MINUS  expr   { Binop($1, Sub, $3) }
  | expr TIMES  expr   { Binop($1, Mul, $3) }
  | expr DIVIDE expr   { Binop($1, Div, $3) }
  | expr LT expr       { Binop($1, Lt, $3) }
  | expr GT expr       { Binop($1, Gt, $3) }
  | expr MOD expr      { Binop($1, Mod, $3) }
  | expr LTE expr      { Binop($1, Lte, $3) }
  | expr GTE expr      { Binop($1, Gte, $3) }
  | expr EQ expr       { Binop($1, Eq, $3) }
  | expr NE expr       { Binop($1, Ne, $3) }
  | expr AND expr      { Binop($1, And, $3) }
  | expr OR expr       { Binop($1, Or, $3) }
  | ID ASSIGN expr     { Assign($1, $3) }
  | ID LBRACKET expr RBRACKET ASSIGN expr { ArrayIndexAssign($1, $3, $6) }  /*
array index assign */
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }  /* function call */
  | LPAREN expr RPAREN { $2 }
  | NOT expr           { Uniop(Not, $2) }

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

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

/* ===================================== */
```

7.7 ast.ml

```
(* open Printf *)

type operator = Add | Sub | Mul | Div | Seq | Lt | Gt | Mod | Lte | Gte
                        | Eq | Ne | And | Or

type unary_operator = Not

type typ = Int | Float | Boolean | Void | String | Array

type expr =
    Liti of int
  | Litf of string
  | Litb of bool
  | Lits of string
  | LitArray of typ * expr
  | Assign of string * expr
  | ArrayIndexAssign of string * expr * expr
  | ArrayIndexAccess of string * expr
  | Id of string
  | Binop of expr * operator * expr
  | Uniop of unary_operator * expr
  | Call of string * expr list
  | Noexpr


type stmt =
    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 bind = typ * string * expr

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

type program = bind list * func_decl list * stmt list



(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mul -> "*"
  | Div -> "/"
  | Eq -> "=="
  | Ne -> "!="
  | Lt -> "<"
  | Lte -> "<="
  | Gt -> ">"
  | Gte -> ">="
  | And -> "&&"
  | Or -> "||"
  | Mod -> "%"

let string_of_typ = function
    Int -> "int"
  | Boolean -> "boolean"
  | Float -> "float"
  | Void -> "void"
  | String -> "string"
  | Array -> "array"

let string_of_uop = function
    Not -> "not"

let rec string_of_expr = function
    Liti(l) -> string_of_int l
  | Litf(l) -> l (* sprintf "%f" l *)
  | Litb(true) -> "true"
  | Litb(false) -> "false"
  | Lits(l) -> l
  | LitArray(t, size) -> "array of type " ^ string_of_typ t ^ " and size " ^ string_of_expr size
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Uniop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | ArrayIndexAssign(s, i, e) -> "array assignment of " ^ string_of_expr e ^ " at index " ^
string_of_expr i ^ " for array " ^ s
  | ArrayIndexAccess(s, i) -> "array " ^ s ^ "at index " ^ string_of_expr i
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
```

```
 | Noexpr -> ""

let rec string_of_stmt = function
    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, e) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_vdecl 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, statements) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs) ^ "\n" ^
  String.concat "\n" (List.map string_of_stmt statements)
```

7.8 sast.ml

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

(* open Printf *)

type sexpr = typ * sx
and sx =
   SLiti of int
  | SLitf of string
  | SLitb of bool
  | SLits of string
  | SLitArray of typ * sexpr
  | SAssign of string * sexpr
  | SArrayIndexAssign of string * sexpr * sexpr
  | SArrayIndexAccess of string * sexpr
  | SId of string
  | SBinop of sexpr * operator * sexpr
  | SUniop of unary_operator * 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 sbind = typ * string * sexpr

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

type sprogram = sbind list * sfunc_decl list * sstmt list


(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SLiti(l) -> string_of_int l
  | SLitf(l) -> l
  | SLitb(true) -> "True"
  | SLitb(false) -> "False"
  | SLits(l) -> l
  | SLitArray(t, size) -> "array of type " ^ string_of_typ t
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUniop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SArrayIndexAssign(s, i, e) -> "array assignment of " ^ string_of_sexpr e ^ " at index " ^
string_of_sexpr i ^ " for array " ^ s
  | SArrayIndexAccess(s, i) -> "array " ^ s ^ "at index " ^ string_of_sexpr i
  | 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_vdecl (t, id, e) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map string_of_vdecl 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, statements) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs) ^ "\n" ^
  String.concat "\n" (List.map string_of_stmt statements)
```

7.9 semant.ml

```
(* Semantic checking for the ComPyled 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, statements) =


  (* 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, e) -> 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;


  (* drop the expression "e" from being stored in the symbol table *)
  let symbols = List.fold_left (fun m (ty, name, e) -> StringMap.add name ty m)
              StringMap.empty globals
  in

  let global_arrays = List.find_all (fun (ty, name, e) -> ty = Array) globals

in

  let array_symbols = List.fold_left (fun m (ty, name, e) -> match e with
      LitArray(t, size) -> StringMap.add name t m)
              StringMap.empty global_arrays
  in


(* Return a variable from our temp symbol table *)
  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in

  (* 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



  let rec expr = function
     Liti l -> (Int, SLiti l)
    | Litf l -> (Float, SLitf l)
    | Litb l  -> (Boolean, SLitb l)
    | Lits l  -> (String, SLits l)
    | Noexpr     -> (Void, SNoexpr)
    | LitArray(t, size) ->
       let sz = expr size in
       (Array, SLitArray(t, sz))
    | Binop(e1, op, e2) as e ->
       let (t1, e1') = expr e1
       and (t2, e2') = expr 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 | Mul | Div | Mod when same && t1 = Int   -> Int
       | Add | Sub | Mul | Div  when same && t1 = Float -> Float
       | Eq | Ne           when same              -> Boolean
       | Lt | Lte | Gt | Gte
                 when same && (t1 = Int || t1 = Float) -> Boolean
       | And | Or when same && t1 = Boolean -> Boolean
       | _ -> 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')))
    | Uniop(op, e) as ex ->
       let (t, e') = expr e in
       let ty = match op with
         Not when t = Boolean -> Boolean
       | _ -> raise (Failure ("illegal unary operator " ^
                          string_of_uop op ^ string_of_typ t ^
                          " in " ^ string_of_expr ex))
       in (ty, SUniop(op, (t, e')))
    | _ -> raise (
     Failure ("Only literals can be initialized with variables. Otherwise you must declare first") )

in
   (***** CHECK expressions of global variables ****)
  (* Return a semantically-checked expression, i.e., with a type *)
let check_globals global =

  let return_checked_global (t, s, e) =
      match e with
        | Noexpr     -> (t, s, expr e)
        | _ ->
          let lt = type_of_identifier s
          and (rt, e') = expr e in
          let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
            string_of_typ rt ^ " in " ^ string_of_expr e
          in check_assign lt rt err;
      (t, s, expr e)

    in return_checked_global global

  in

  (* Collect function declarations for built-in functions: no bodies *)
    let built_in_decls1 =
      let add_bind map (name, ty) = StringMap.add name {
        typ = Void;
        fname = name;
        formals = [(ty, "x", Noexpr)];
        locals = [];
        body = [] } map
      in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                                   ("printb", Boolean);
                                                   ("printf", Float);
                                                   ("prints", String);]
    in

    let built_in_decls2 =
      let add_bind map (name, ty) = StringMap.add name {
        typ = String;
        fname = name;
        formals = [(ty, "x", Noexpr)];
        locals = [];
        body = [] } map
      in List.fold_left add_bind built_in_decls1 [ ("string_concat", String); ]
    in

    (* let built_in_decls3 =
      let add_bind map (name, ty) = StringMap.add name {
        typ = Boolean;
        fname = name;
    formals = [(ty, "x", Noexpr)];
    locals = [];
    body = [] } map
  in List.fold_left add_bind built_in_decls2 [ ("string_equality", String) ]
in *)

let built_in_decls =
  let add_bind map (name, ty) = StringMap.add name {
    typ = Int;
    fname = name;
    formals = [(ty, "x", Noexpr)];
    locals = [];
    body = [] } map
  in List.fold_left add_bind built_in_decls2 [ ("len", String); ]
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

let main_name = "main"
in

let find_main name =
  try (StringMap.find name function_decls).fname
  with Not_found -> ""
in


(* add the main function. If user defined a function with same name, recurse to
    create one with a name they didn't use.
    Just keep adding a "0" after main's function name until no matches
  *)

 let rec create_main name = match find_main name with
       "" -> StringMap.add name {typ = Int; fname = name; formals = []; locals = []; body = [] }
function_decls
      | _ -> create_main (name ^ "0")
 in
 let function_decls = create_main main_name

 in
 (**** Check Functions ****)


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

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


 (* FOR LOCAL VAR INIT and formal check *)
  (* Build local symbol table to check the types of the initializations *)
   (* drop the expression "e" from being stored in the symbol table*)
   let local_symbols = List.fold_left (fun m (ty, name, e) -> StringMap.add name ty m)
                 StringMap.empty (globals @ func.formals @ func.locals)
   in

   let local_arrays = List.find_all (fun (ty, name, e) -> ty = Array) func.locals

   in

   let local_array_symbols = List.fold_left (fun m (ty, name, e) -> match e with
        LitArray(t, size) -> StringMap.add name t m)
                 StringMap.empty local_arrays
   in
   (* Return a variable from the local symbol table *)
   let type_of_identifier s =
     try StringMap.find s local_symbols
     with Not_found -> raise (Failure ("undeclared identifier " ^ s))
   in

   (* Return a array type from our array symbol table *)
   let type_of_array_identifier s =
     try StringMap.find s local_array_symbols
     with Not_found -> raise (Failure ("undeclared identifier " ^ s))
   in

   let check_print lvaluet rvaluet err =
      lvaluet
   in

    (* Return a semantically-checked expression, i.e., with a type *)
   let rec expr = function
       Liti l -> (Int, SLiti l)
     | Litf l -> (Float, SLitf l)
     | Litb l  -> (Boolean, SLitb l)
     | Lits l  -> (String, SLits l)
     | LitArray(t, size) ->
         let sz = expr size in
         (Array, SLitArray(t, sz))
     | Noexpr     -> (Void, SNoexpr)
     | Id s       -> (type_of_identifier s, SId s)
     | Assign(var, e) as ex ->
         let lt = type_of_identifier var
         and (rt, e') = expr e in
         let rt = if rt = Array then let SArrayIndexAccess(var, (rt_i, idx)) = e' in
type_of_array_identifier var else rt
         in
         let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
           string_of_typ rt ^ " in " ^ string_of_expr ex
         in
         (check_assign lt rt err, SAssign(var, (rt, e')))
     | ArrayIndexAssign(var, idx, e) as ex ->
         let lt = type_of_array_identifier var
         and (rt, e') = expr e
         and (rt_i, indx) = expr idx
         in
         let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
           string_of_typ rt ^ " in " ^ string_of_expr ex
         in
         (check_assign lt rt err, SArrayIndexAssign(var, (rt_i, indx), (rt, e')))
     | ArrayIndexAccess(var, idx) -> let (rt_i, indx) = expr idx in
         (type_of_array_identifier var, SArrayIndexAccess(var, (rt_i, indx)))
     | Binop(e1, op, e2) as e ->
         let (t1, e1') = expr e1
         and (t2, e2') = expr 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 | Mul | Div | Mod when same && t1 = Int   -> Int
         | Add | Sub | Mul | Div  when same && t1 = Float -> Float
         | Add when same && t1 = String -> String
         | Eq | Ne          when same               -> Boolean
         | Lt | Lte | Gt | Gte
                 when same && (t1 = Int || t1 = Float) -> Boolean
         | And | Or when same && t1 = Boolean -> Boolean
         | _ -> 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')))
     | Uniop(op, e) as ex ->
         let (t, e') = expr e in
         let ty = match op with
           Not when t = Boolean -> Boolean
         | _ -> raise (Failure ("illegal unary operator " ^
                          string_of_uop op ^ string_of_typ t ^
                          " in " ^ string_of_expr ex))
         in (ty, SUniop(op, (t, e')))
     | 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 ("wrong number of args "))
         else let check_call (ft, _, _) e =
           let (et, e') = expr e in
           let err = "illegal argument found "
```

```
        (* skip print function *)
        in if fname = "print" then (check_print ft et err, e') else (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_locals local =
  let return_checked_locals (t, s, e) =
    match e with
        | Noexpr     -> (t, s, expr e)
        | _ ->
          let lt = type_of_identifier s
          and (rt, e') = expr e in
          let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
            string_of_typ rt ^ " in " ^ string_of_expr e
          in check_assign lt rt err;
      (t, s, expr e)

     in return_checked_locals local

in

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

  (* Return a semantically-checked statement i.e. containing sexprs *)
  let rec check_stmt = function
      Expr e -> SExpr (expr e)
    | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
    | For(e1, e2, e3, st) ->
        SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
    | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
    | Return e -> let (t, e') = expr 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))
    | Block sl ->
```

```
        let rec check_stmt_list = function
            [Return _ as s] -> [check_stmt s]
          | Return _ :: _   -> raise (Failure "nothing may follow a return")
          | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
          | s :: ss         -> check_stmt s :: check_stmt_list ss
          | []              -> []
        in SBlock(check_stmt_list sl)


    (* Need to return semantically checked expressions for locals *)

    in (* body of check_function *)
    { styp = func.typ;
      sfname = func.fname;
      sformals = (List.map check_locals func.formals);
      slocals  = (List.map check_locals func.locals);
      sbody = match check_stmt (Block func.body) with
  SBlock(sl) -> sl
      | _ -> raise (Failure ("internal error: block didn't become a block?"))
    }


in

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


  let check_statement statements =

    (* Return a variable from the global symbol table *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in

    (* Return a array type from the global array symbol table *)
    let type_of_array_identifier s =
      try StringMap.find s array_symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in

  let check_print lvaluet rvaluet err =
    lvaluet
  in
  (* Return a semantically-checked expression, i.e., with a type *)
  let rec expr = function
      Liti l -> (Int, SLiti l)
    | Litf l -> (Float, SLitf l)
    | Litb l  -> (Boolean, SLitb l)
    | Lits l  -> (String, SLits l)
    | LitArray(t, size) ->
        let sz = expr size in
        (Array, SLitArray(t, sz))
    | Noexpr     -> (Void, SNoexpr)
    | Id s       -> (type_of_identifier s, SId s)
    | Assign(var, e) as ex ->
        let lt = type_of_identifier var
        and (rt, e') = expr e in
        let rt = if rt = Array then let SArrayIndexAccess(var, (rt_i, idx)) = e' in
type_of_array_identifier var else rt
        in
        let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
          string_of_typ rt ^ " in " ^ string_of_expr ex
        in
        (check_assign lt rt err, SAssign(var, (rt, e')))
    | ArrayIndexAssign(var, idx, e) as ex ->
        let lt = type_of_array_identifier var
        and (rt, e') = expr e
        and (rt_i, indx) = expr idx
        in
        let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
          string_of_typ rt ^ " in " ^ string_of_expr ex
        in
        (check_assign lt rt err, SArrayIndexAssign(var, (rt_i, indx), (rt, e')))
    | ArrayIndexAccess(var, idx) -> let (rt_i, indx) = expr idx in
        (type_of_array_identifier var, SArrayIndexAccess(var, (rt_i, indx)))
    | Binop(e1, op, e2) as e ->
        let (t1, e1') = expr e1
        and (t2, e2') = expr 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 | Mul | Div | Mod when same && t1 = Int   -> Int
      | Add | Sub | Mul | Div  when same && t1 = Float -> Float
      | Add when same && t1 = String -> String
      | Eq | Ne            when same               -> Boolean
      | Lt | Lte | Gt | Gte
                when same && (t1 = Int || t1 = Float) -> Boolean
      | And | Or when same && t1 = Boolean -> Boolean
      | _ -> 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')))
  | Uniop(op, e) as ex ->
      let (t, e') = expr e in
      let ty = match op with
        Not when t = Boolean -> Boolean
      | _ -> raise (Failure ("illegal unary operator " ^
                           string_of_uop op ^ string_of_typ t ^
                           " in " ^ string_of_expr ex))
      in (ty, SUniop(op, (t, e')))
  | 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 ("wrong number of args "))
      else let check_call (ft, _, _) e =
        let (et, e') = expr e in
        let err = "illegal argument found "
        (* skip print function *)
        in if fname = "print" then (check_print ft et err, e') else (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 e =
  let (t', e') = expr e
  and err = "expected Boolean expression in " ^ string_of_expr e
  in if t' != Boolean then raise (Failure err) else (t', e')
 in

 (* Return a semantically-checked statement i.e. containing sexprs *)
 let rec check_stmt = function
    Expr e -> SExpr (expr e)
  | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
  | For(e1, e2, e3, st) ->
     SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
  | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
  | Return e -> let (t, e') = expr e in
     SReturn (t, e')
  | Block sl ->
     let rec check_stmt_list = function
         [Return _ as s] -> [check_stmt s]
       | Return _ :: _   -> raise (Failure "nothing may follow a return")
       | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
       | s :: ss         -> check_stmt s :: check_stmt_list ss
       | []              -> []
     in SBlock(check_stmt_list sl)

 in

 match check_stmt (Block statements) with
    SBlock(sl) -> sl
  | _ -> raise (Failure ("internal error: block didn't become a block?"))


 in


(* Return a variable from the global symbol table *)
 let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found -> raise (Failure ("undeclared identifier " ^ s))
 in

 (* Return a array type from the global array symbol table *)
 let type_of_array_identifier s =
  try StringMap.find s array_symbols
  with Not_found -> raise (Failure ("undeclared identifier " ^ s))
 in
```

```
  let check_print lvaluet rvaluet err =
    lvaluet
  in


   (* Return a semantically-checked expression, i.e., with a type *)
  let rec expr = function
      Liti l -> (Int, SLiti l)
    | Litf l -> (Float, SLitf l)
    | Litb l  -> (Boolean, SLitb l)
    | Lits l  -> (String, SLits l)
    | LitArray(t, size) ->
        let sz = expr size in
        (Array, SLitArray(t, sz))
    | Noexpr     -> (Void, SNoexpr)
    | Id s       -> (type_of_identifier s, SId s)
    | Assign(var, e) as ex ->
        let lt = type_of_identifier var
        and (rt, e') = expr e in
        let rt = if rt = Array then let SArrayIndexAccess(var, (rt_i, idx)) = e' in
type_of_array_identifier var else rt
        in
        let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
          string_of_typ rt ^ " in " ^ string_of_expr ex
        in
        (check_assign lt rt err, SAssign(var, (rt, e')))
    | ArrayIndexAssign(var, idx, e) as ex ->
        let lt = type_of_array_identifier var
        and (rt, e') = expr e
        and (rt_i, indx) = expr idx
        in
        let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
          string_of_typ rt ^ " in " ^ string_of_expr ex
        in
        (check_assign lt rt err, SArrayIndexAssign(var, (rt_i, indx), (rt, e')))
    | ArrayIndexAccess(var, idx) -> let (rt_i, indx) = expr idx in
        (type_of_array_identifier var, SArrayIndexAccess(var, (rt_i, indx)))
    | Binop(e1, op, e2) as e ->
        let (t1, e1') = expr e1
        and (t2, e2') = expr 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 | Mul | Div | Mod when same && t1 = Int   -> Int
      | Add | Sub | Mul | Div  when same && t1 = Float -> Float
      | Add when same && t1 = String -> String
      | Eq | Ne           when same              -> Boolean
      | Lt | Lte | Gt | Gte
               when same && (t1 = Int || t1 = Float) -> Boolean
      | And | Or when same && t1 = Boolean -> Boolean
      | _ -> 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')))
  | Uniop(op, e) as ex ->
      let (t, e') = expr e in
      let ty = match op with
        Not when t = Boolean -> Boolean
      | _ -> raise (Failure ("illegal unary operator " ^
                        string_of_uop op ^ string_of_typ t ^
                        " in " ^ string_of_expr ex))
      in (ty, SUniop(op, (t, e')))
  | 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 ("wrong number of args "))
      else let check_call (ft, _, _) e =
        let (et, e') = expr e in
        let err = "illegal argument found "
        (* skip print function *)
        in if fname = "print" then (check_print ft et err, e') else (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 e =
  let (t', e') = expr e
  and err = "expected Boolean expression in " ^ string_of_expr e
```

```
    in if t' != Boolean then raise (Failure err) else (t', e')
  in

  (* Return a semantically-checked statement i.e. containing sexprs *)
  let rec check_stmt = function
      Expr e -> SExpr (expr e)
    | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
    | For(e1, e2, e3, st) ->
        SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
    | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
    | Return e -> let (t, e') = expr e in
        SReturn (t, e')
    | Block sl ->
        let rec check_stmt_list = function
            [Return _ as s] -> [check_stmt s]
          | Return _ :: _   -> raise (Failure "nothing may follow a return")
          | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
          | s :: ss         -> check_stmt s :: check_stmt_list ss
          | []              -> []
        in SBlock(check_stmt_list sl)


 in (List.map check_globals globals, List.map check_function functions, check_statement
statements)
```

7.10 codegen.ml

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

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
(* context = the thing we need to pass to certain LLVM functions
internally: some C++ class
*)

let translate (globals, functions, statements) =
  let context    = L.global_context () in

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


  (* Get types from the context *)
  (* llvm only supports primitive types *)
  let i32_t      = L.i32_type      context  (* 32-bit int type *)
  and i8_t       = L.i8_type       context  (* caracters *)
  and i1_t       = L.i1_type       context  (* boolean type *)
  and float_t    = L.double_type   context  (* double/float type *)
  and void_t     = L.void_type     context  (* void type *)
  and string_t   = L.pointer_type  (L.i8_type context)    (* pointer type to char *)
  (* and array_t    = L.array_type     (L.i32_type context) *)
  in

  (* Return the LLVM type for a complyed type *)
  let ltype_of_typ = function
      A.Int   -> i32_t
    | A.Boolean  -> i1_t
    | A.Float -> float_t
    | A.Void  -> void_t
    | A.String -> string_t
    (* | A.Array -> array_t *)
  in


  (* Declaring external functions *)
  (* create a link to the C library's "printf" *)
  let printf_t : L.lltype =
      (* the [| and |] indicates an Ocaml array*)
      L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
    (* below LLVM's connection to a built-in function *)
  let printf_func : L.llvalue =
      L.declare_function "printf" printf_t the_module in
```

```
(* BUILTIN BEGIN *)

let string_concat_t : L.lltype =
  L.function_type string_t [| string_t; string_t |] in
let string_concat_f : L.llvalue =
  L.declare_function "string_concat" string_concat_t the_module in

(* let string_equality_t : L.lltype =
  L.function_type string_t [| string_t; string_t |] in
let string_equality_f : L.llvalue =
  L.declare_function "string_equality" string_equality_t the_module in *)

let len_t : L.lltype =
  L.function_type i32_t [| string_t |] in
let len_f : L.llvalue =
  L.declare_function "len" len_t the_module in

(* BUILTIN END *)

(* create fake main function *)
let main_t : L.lltype =
    (* the [| and |] indicates an Ocaml array*)
    L.var_arg_function_type i32_t [| L.pointer_type i8_t |] 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,_, e) -> 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



(* Generate a name for a simulated main function. Recursively give it a new name in
   case the user names one of their functions as "main".
*)
```

```
  let main_name = "main" in


   let test_main_name main_name =
     try
       let (_,the_function) = StringMap.find main_name function_decls
       in the_function.sfname
     with Not_found -> main_name

  in

  let rec generate_main_name name = (* match test_main_name name with *)
    if test_main_name name == name then name
    else generate_main_name (name ^ "0")


 in let main_name = generate_main_name main_name in

 let function_decls = StringMap.add main_name (L.define_function main_name main_t
the_module, ({styp = Int; sfname = main_name; sformals = []; slocals = []; sbody = [] }))
function_decls



 in

 (* create a main function builder hidden from the user to wrap the entire script in *)
 (* Needs to occur outside of the build_statement function *)
 (* Need to find the main function with the most zeros at the end *)
 let (the_function, _) = StringMap.find main_name function_decls in
 let builder = L.builder_at_end context (L.entry_block the_function) in



(********* THIS EXPR BUILDER IS SOLELY FOR VARIABLE INITIALIZATION!!!!
******)
let rec expr builder ((_, e) : sexpr) = match e with
       SLiti i  -> L.const_int i32_t i
     | SLitb b  -> L.const_int i1_t (if b then 1 else 0)
     | SLitf l -> L.const_float_of_string float_t l
     | SLits s -> L.build_global_stringptr s "str" builder
     | SNoexpr     -> L.const_int i32_t 0
| SBinop ((A.Float,_ ) as e1, op, e2) ->
  let e1' = expr builder e1
  and e2' = expr builder e2 in
    (match op with
      A.Add     -> L.build_fadd
    | A.Sub     -> L.build_fsub
    | A.Mul     -> L.build_fmul
    | A.Div     -> L.build_fdiv
    | A.Mod     -> L.build_urem
    | A.Eq   -> L.build_fcmp L.Fcmp.Oeq
    | A.Ne     -> L.build_fcmp L.Fcmp.One
    | A.Lt    -> L.build_fcmp L.Fcmp.Olt
    | A.Lte     -> L.build_fcmp L.Fcmp.Ole
    | A.Gt -> L.build_fcmp L.Fcmp.Ogt
    | A.Gte     -> L.build_fcmp L.Fcmp.Oge
    | A.And | A.Or ->
        raise (Failure "internal error: semant should have rejected and/or on float")
    ) e1' e2' "tmp" builder
| SBinop ((A.String,_ ) as e1, op, e2) ->
  let e1' = expr builder e1
  and e2' = expr builder e2 in
  (match op with
    A.Add  -> L.build_call string_concat_f [| e1'; e2' |] "string_concat" builder)
   (* | A.Eq   -> L.build_call string_equality_f [| e1'; e2' |] "string_equality" builder) *)
| SBinop (e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add     -> L.build_add
    | A.Sub     -> L.build_sub
    | A.Mul     -> L.build_mul
    | A.Div     -> L.build_sdiv
    | A.Mod     -> L.build_urem
    | A.And     -> L.build_and
    | A.Or      -> L.build_or
    | A.Eq   -> L.build_icmp L.Icmp.Eq
    | A.Ne     -> L.build_icmp L.Icmp.Ne
    | A.Lt    -> L.build_icmp L.Icmp.Slt
    | A.Lte     -> L.build_icmp L.Icmp.Sle
    | A.Gt -> L.build_icmp L.Icmp.Sgt
    | A.Gte     -> L.build_icmp L.Icmp.Sge
    ) e1' e2' "tmp" builder
```

```
      | SUniop(op, ((t, _) as e)) ->
          let e' = expr builder e in
        (match op with
            A.Not -> L.build_not) e' "tmp" builder
      | SCall ("len", [e]) ->
        let e' = expr builder e in
        L.build_call len_f  [| e' |] "len" builder
  in
 (* Create a map of global variables after creating each *)
 let global_vars : L.llvalue StringMap.t =
  let global_var m (t, n, se) =
    let init = match se with
      (A.Void, _) ->  (
          match t with
            A.Float -> L.build_alloca (ltype_of_typ t) n builder;
          | A.Int -> L.build_alloca (ltype_of_typ t) n builder;
          | A.Boolean -> L.build_alloca (ltype_of_typ t) n builder;
          | A.String -> L.build_alloca (ltype_of_typ t) n builder;
         )
      | (A.Array, _) -> (match snd se with
                SLitArray(t, size) -> L.build_array_alloca (ltype_of_typ t) (expr builder size) n
builder
              )
              (* L.build_array_malloc (ltype_of_typ t) (L.const_int i32_t size) n builder ) *)
      | _ -> L.build_alloca (ltype_of_typ t) n builder
    in let (ty, s) = se
    in if t <> A.Array && ty <> A.Void then ignore (L.build_store (expr builder se) init
builder)
      else if ty = A.Void && (t == A.Int || t = A.Boolean) then ignore (L.build_store
(L.const_int (ltype_of_typ t) 0) init builder)
      else if ty = A.Void && t == A.Float then ignore (L.build_store (L.const_float
(ltype_of_typ t) 0.0) init builder)
      else if ty = A.Void && t == A.String then ignore (L.build_store (L.const_pointer_null
(ltype_of_typ t)) init builder);
    StringMap.add n init m
  in

  List.fold_left global_var StringMap.empty globals in
```

```
(* in ignore(L.build_store (expr builder se) init builder); StringMap.add n (init) m *)
(* Create a map of global variables after creating each *)
let global_var_types : A.typ StringMap.t =
  let global_var m (t, n, se) =
    let atype = t
    in StringMap.add n t m in
  List.fold_left global_var StringMap.empty globals in


let global_arrays = List.find_all (fun (ty, name, e) -> ty = A.Array) globals

in

  let array_symbols = List.fold_left (fun m (ty, name, e) -> match e with
      (_, SLitArray(t, size)) -> StringMap.add name t m)
              StringMap.empty global_arrays
  in




(******** BUILD FUNCTIONS ************)

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

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


  (********* THIS EXPR BUILDER IS SOLELY FOR INITIALIZATION!!!! ******)
  (* Construct code for an expression in the INITIALIZATION; return its value *)
  let rec expr ((_, e) : sexpr) = match e with
      SLiti i  -> L.const_int i32_t i
    | SLitb b  -> L.const_int i1_t (if b then 1 else 0)
    | SLitf l -> L.const_float_of_string float_t l
    | SLits s -> L.build_global_stringptr s "str" f_builder
    | SNoexpr     -> L.const_int i32_t 0
    | SAssign (s, e) -> expr e
```

```
 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, se) p =
       L.set_value_name n p;
     let local = L.build_alloca (ltype_of_typ t) n f_builder
         in ignore (L.build_store p local f_builder);
         StringMap.add n local m

     (* Allocate space for any locally declared variables and add the
      * resulting registers to our map *)
     and add_local m (t, n, se) =
       let local_var = match se with
           (A.Void, _) -> L.build_alloca (ltype_of_typ t) n f_builder
         | (A.Array, _) -> (match snd se with
                 SLitArray(ty, size) -> L.build_array_alloca (ltype_of_typ ty) (expr size) n f_builder
               )
         | _ -> L.build_alloca (ltype_of_typ t) n f_builder
       in let (ty, s) = se in
       if t <> A.Array && (ty <> A.Void) then ignore (L.build_store (expr se) local_var
f_builder);
       StringMap.add n local_var m

     in

     let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals
         (Array.to_list (L.params the_function)) in
     List.fold_left add_local formals fdecl.slocals
   in

   (* Create a map of global variables after creating each *)
 let local_var_types : A.typ StringMap.t =
   let local_var m (t, n, se) =
     let atype = t
     in StringMap.add n t m in
   List.fold_left local_var StringMap.empty fdecl.slocals in
  (* Return the value for a variable or formal argument.
     Check local names first, then global names *)
```

```
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars
    in

    let find_type n = try StringMap.find n local_var_types
                   with Not_found -> raise (Failure "local variable type not found")
    in


    (* Construct code for an expression; return its value *)
    (* An expression in LLVM always turns into code in a single basic block (not true for stmts)
  *)
    (* build instructions in the given builder that evaluate the expr; return the expr's value *)
    let rec expr f_builder ((_, e) : sexpr) = match e with
        SLiti i  -> L.const_int i32_t i
      | SLitb b  -> L.const_int i1_t (if b then 1 else 0)
      | SLitf l -> L.const_float_of_string float_t l
      | SLits s -> L.build_global_stringptr s "str" f_builder
      | SNoexpr     -> L.const_int i32_t 0
      | SId s       -> L.build_load (lookup s) s f_builder
      | SAssign (s, e) -> let e' = expr f_builder e in
                        ignore(L.build_store e' (lookup s) f_builder); e'
      | SArrayIndexAssign (s, idx, e) ->
                        let e' = expr f_builder e and
                        indx = expr f_builder idx
                        in
                        let element_ptr = L.build_gep (lookup s) [| indx |] "" f_builder and
                        e' = e'
                       in
                        ignore(L.build_store e' element_ptr f_builder); e'
      | SArrayIndexAccess (s, idx) ->
                  let indx = expr f_builder idx in
                  let element_ptr = L.build_load(L.build_gep (lookup s) [| indx |] "" f_builder) ""
  f_builder
                  in element_ptr
      | SBinop ((A.Float,_ ) as e1, op, e2) ->
        let e1' = expr f_builder e1
        and e2' = expr f_builder e2 in
         (match op with
            A.Add     -> L.build_fadd
          | A.Sub     -> L.build_fsub
          | A.Mul    -> L.build_fmul
          | A.Div     -> L.build_fdiv
          | A.Mod   -> L.build_urem
          | A.Eq   -> L.build_fcmp L.Fcmp.Oeq
          | A.Ne     -> L.build_fcmp L.Fcmp.One
          | A.Lt    -> L.build_fcmp L.Fcmp.Olt
          | A.Lte     -> L.build_fcmp L.Fcmp.Ole
          | A.Gt -> L.build_fcmp L.Fcmp.Ogt
          | A.Gte     -> L.build_fcmp L.Fcmp.Oge
          | A.And | A.Or ->
            raise (Failure "internal error: semant should have rejected and/or on float")
          ) e1' e2' "tmp" f_builder
            | SBinop (e1, op, e2) ->
          let e1' = expr f_builder e1
          and e2' = expr f_builder e2 in
          (match op with
            A.Add     -> L.build_add
          | A.Sub     -> L.build_sub
          | A.Mul    -> L.build_mul
          | A.Div     -> L.build_sdiv
          | A.Mod     -> L.build_urem
          | A.And     -> L.build_and
          | A.Or      -> L.build_or
          | A.Eq   -> L.build_icmp L.Icmp.Eq
          | A.Ne     -> L.build_icmp L.Icmp.Ne
          | A.Lt    -> L.build_icmp L.Icmp.Slt
          | A.Lte     -> L.build_icmp L.Icmp.Sle
          | A.Gt -> L.build_icmp L.Icmp.Sgt
          | A.Gte     -> L.build_icmp L.Icmp.Sge
         ) e1' e2' "tmp" f_builder
      | SBinop ((A.String,_ ) as e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
          A.Add  -> L.build_call string_concat_f [| e1'; e2' |] "string_concat" builder)
         (* | A.Eq   -> L.build_call string_equality_f [| e1'; e2' |] "string_equality" builder) *)
      | SUniop(op, ((t, _) as e)) ->
         let e' = expr f_builder e in
        (match op with
          A.Not -> L.build_not) e' "tmp" f_builder
      | SCall ("print", [e]) ->
         let e' = expr f_builder e in
         ( match e with
                (_, SLiti i)  -> L.build_call printf_func [| int_format_str ; e' |] "printf" f_builder
              | (_, SLitb b)  -> L.build_call printf_func [| int_format_str ; e' |] "printf" f_builder
              | (_, SLitf l) -> L.build_call printf_func [| float_format_str ; e' |] "printf" f_builder
              | (_, SLits s) -> L.build_call printf_func [| string_format_str ; e' |] "printf" f_builder
              | (_, SId s) ->
                 ( match find_type s with
                     A.Int     -> L.build_call printf_func [| int_format_str ; e' |] "printf" f_builder
                   | A.Boolean      -> L.build_call printf_func [| int_format_str ; e' |] "printf" f_builder
                   | A.Float   -> L.build_call printf_func [| float_format_str ; e' |] "printf" f_builder
                   | A.String  -> L.build_call printf_func [| string_format_str ; e' |] "printf" f_builder
                   | _ -> raise (Failure "invalid argument called on the print function")
                 )
              | (_, SBinop ((A.Float,_ ) as e1, op, e2)) ->  L.build_call printf_func [| float_format_str ;
e' |] "printf" builder
          | (_, SBinop (e1, op, e2)) ->  L.build_call printf_func [| int_format_str ; e' |] "printf"
f_builder
          | (_, SCall(f, args)) -> L.build_call printf_func [| int_format_str ; e' |] "printf" f_builder
          | (_,_) ->  raise (Failure "invalid argument called on the print function")
        )
    | SCall ("len", [e]) ->
      let e' = expr builder e in
      L.build_call len_f  [| e' |] "len" builder
    | SCall (f, args) ->
       let (fdef, fdecl) = StringMap.find f function_decls in
         let llargs = List.rev (List.map (expr f_builder) (List.rev args)) in
         let result = (match fdecl.styp with
            A.Void -> ""
          | _ -> f ^ "_result") in
         L.build_call fdef (Array.of_list llargs) result f_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 f_builder instr =
    match L.block_terminator (L.insertion_block f_builder) with
       Some _ -> ()
     | None -> ignore (instr f_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 f_builder = function
   SBlock sl -> List.fold_left stmt f_builder sl
  | SExpr e -> ignore(expr f_builder e); f_builder
  | SReturn e -> ignore(match fdecl.styp with
                      (* Special "return nothing" instr *)
                      A.Void -> L.build_ret_void f_builder
                      (* Build return statement *)
                     | _ -> L.build_ret (expr f_builder e) f_builder );
             f_builder
  | SIf (predicate, then_stmt, else_stmt) ->
   let bool_val = expr f_builder 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 (stmt (L.builder_at_end context then_bb) then_stmt)
     build_br_merge;

   let else_bb = L.append_block context "else" the_function in
   add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
     build_br_merge;

     ignore(L.build_cond_br bool_val then_bb else_bb f_builder);
     L.builder_at_end context merge_bb
  | SWhile (predicate, body) ->
    let pred_bb = L.append_block context "while" the_function in
    ignore(L.build_br pred_bb f_builder);

    let body_bb = L.append_block context "while_body" the_function in
    add_terminal (stmt (L.builder_at_end context body_bb) body)
      (L.build_br pred_bb);

    let pred_builder = L.builder_at_end context pred_bb in
    let bool_val = expr pred_builder 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
  (* Implement for loops as while loops *)
  | SFor (e1, e2, e3, body) -> stmt f_builder
      ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
      in

       (* Build the code for each statement *)
      let f_builder = stmt f_builder (SBlock fdecl.sbody) in


      (* Add a return if the last block falls off the end *)
      add_terminal f_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
     (****** BUILD STATEMENTS THAT EXIST OUTSIDE OF FUNCTION DEFINITONS
*********)

 let build_statements statements =

   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 string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
 in


  (* Return the value for a variable or formal argument.
    Check local names first, then global names *)
  let lookup n = try StringMap.find n global_vars
              with Not_found -> raise (Failure "global variable not found")
  in

  let find_array_type n = try StringMap.find n array_symbols
              with Not_found -> raise (Failure "array type not found")
  in


  let find_type n = try StringMap.find n global_var_types
              with Not_found -> raise (Failure "global variable type not found")
  in


  (* Construct code for an expression; return its value *)
```

```
    (* An expression in LLVM always turns into code in a single basic block (not true for stmts)
*)
    (* build instructions in the given builder that evaluate the expr; return the expr's value *)
    let rec expr builder ((_, e) : sexpr) = match e with
        SLiti i  -> L.const_int i32_t i
      | SLitb b  -> L.const_int i1_t (if b then 1 else 0)
      | SLitf l -> L.const_float_of_string float_t l
      | SLits s -> L.build_global_stringptr s "str" builder
      | SNoexpr     -> L.const_int i32_t 0
      | SId s       -> L.build_load (lookup s) s builder
      | SAssign (s, e) -> let e' = expr builder e in
                        ignore(L.build_store e' (lookup s) builder); e'
      | SAssign (s, (ty, SArrayIndexAssign (nam, idx, e))) ->
                        let e' = expr builder e and
                        indx = expr builder idx in
                        let element_ptr = L.build_gep (lookup nam) [| indx |] "" builder
                        in
                        ignore(L.build_store e' element_ptr builder); e'
      | SArrayIndexAssign (s, idx, e) ->
                        let e' = expr builder e and
                        indx = expr builder idx
                        in
                        let element_ptr = L.build_gep (lookup s) [| indx |] "" builder and
                        e' = e'
                      in  ignore(L.build_store e' element_ptr builder); e'
      | SArrayIndexAccess (s, idx) ->
                  let indx = expr builder idx in
                  let element_ptr = L.build_load(L.build_gep (lookup s) [| indx |] "" builder) ""
builder
                  in element_ptr
      | SBinop ((A.Float,_ ) as e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
          (match op with
            A.Add     -> L.build_fadd
          | A.Sub     -> L.build_fsub
          | A.Mul    -> L.build_fmul
          | A.Div     -> L.build_fdiv
          | A.Mod     -> L.build_urem
          | A.Eq   -> L.build_fcmp L.Fcmp.Oeq
          | A.Ne     -> L.build_fcmp L.Fcmp.One
          | A.Lt    -> L.build_fcmp L.Fcmp.Olt
          | A.Lte     -> L.build_fcmp L.Fcmp.Ole
          | A.Gt -> L.build_fcmp L.Fcmp.Ogt
          | A.Gte     -> L.build_fcmp L.Fcmp.Oge
          | A.And | A.Or ->
              raise (Failure "internal error: semant should have rejected and/or on float")
          ) e1' e2' "tmp" builder
      | SBinop ((A.String,_ ) as e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
          A.Add  -> L.build_call string_concat_f [| e1'; e2' |] "string_concat" builder)
         (* | A.Eq   -> L.build_call string_equality_f [| e1'; e2' |] "string_equality" builder) *)
      | SBinop (e1, op, e2) ->
          let e1' = expr builder e1
          and e2' = expr builder e2 in
          (match op with
            A.Add     -> L.build_add
          | A.Sub     -> L.build_sub
          | A.Mul     -> L.build_mul
          | A.Div     -> L.build_sdiv
          | A.Mod     -> L.build_urem
          | A.And     -> L.build_and
          | A.Or      -> L.build_or
          | A.Eq   -> L.build_icmp L.Icmp.Eq
          | A.Ne     -> L.build_icmp L.Icmp.Ne
          | A.Lt    -> L.build_icmp L.Icmp.Slt
          | A.Lte     -> L.build_icmp L.Icmp.Sle
          | A.Gt -> L.build_icmp L.Icmp.Sgt
          | A.Gte     -> L.build_icmp L.Icmp.Sge
          ) e1' e2' "tmp" builder
      | SUniop(op, ((t, _) as e)) ->
          let e' = expr builder e in
        (match op with
          A.Not -> L.build_not) e' "tmp" builder
      | SCall ("print", [e]) ->
          let e' = expr builder e in
          ( match e with
            (_, SLiti i)  -> L.build_call printf_func [| int_format_str ; e' |] "printf" builder
          | (_, SLitb b)  -> L.build_call printf_func [| int_format_str ; e' |] "printf" builder
          | (_, SLitf l) -> L.build_call printf_func [| float_format_str ; e' |] "printf" builder
          | (_, SLits s) -> L.build_call printf_func [| string_format_str ; e' |] "printf" builder
          | (_, SId s) ->
```

```
            ( match find_type s with
                A.Int     -> L.build_call printf_func [| int_format_str ; e' |] "printf" builder
              | A.Boolean      -> L.build_call printf_func [| int_format_str ; e' |] "printf" builder
              | A.Float   -> L.build_call printf_func [| float_format_str ; e' |] "printf" builder
              | A.String  -> L.build_call printf_func [| string_format_str ; e' |] "printf" builder
              | _ -> raise (Failure "invalid argument called on the print function")
            )
          | (A.Int, SArrayIndexAccess (s, i)) -> L.build_call printf_func [| int_format_str ; e' |]
"printf" builder
          | (A.Float, SArrayIndexAccess (s, i)) -> L.build_call printf_func [| float_format_str ; e'
|] "printf" builder
          | (A.String, SArrayIndexAccess (s, i)) -> L.build_call printf_func [| string_format_str ;
e' |] "printf" builder
          | (_, SBinop ((A.Float,_ ) as e1, op, e2)) ->  L.build_call printf_func [| float_format_str ;
e' |] "printf" builder
          | (_, SBinop (e1, op, e2)) ->  L.build_call printf_func [| int_format_str ; e' |] "printf"
builder
          | (_, SCall(f, args)) -> L.build_call printf_func [| int_format_str ; e' |] "printf" builder
          | (_,_) ->  raise (Failure "invalid argument called on the print function")
        )
      | SCall ("len", [e]) ->
        let e' = expr builder e in
        L.build_call len_f  [| e' |] "len" builder
      | SCall (f, args) ->
         let (fdef, fdecl) = StringMap.find f function_decls in
           let llargs = List.rev (List.map (expr builder) (List.rev args)) in
           let result = (match fdecl.styp with
               A.Void -> ""
             | _ -> f ^ "_result") in
           L.build_call fdef (Array.of_list llargs) result 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 = function
   SBlock sl -> List.fold_left stmt builder sl
   | SExpr e -> ignore(expr builder e); builder
   | SReturn e -> raise (Failure "Return should not be specified without a function defintion")
   | SIf (predicate, then_stmt, else_stmt) ->
    let bool_val = expr builder 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 (stmt (L.builder_at_end context then_bb) then_stmt)
      build_br_merge;

    let else_bb = L.append_block context "else" the_function in
    add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
      build_br_merge;

      ignore(L.build_cond_br bool_val then_bb else_bb builder);
      L.builder_at_end context merge_bb
   | 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 (stmt (L.builder_at_end context body_bb) body)
       (L.build_br pred_bb);

     let pred_builder = L.builder_at_end context pred_bb in
     let bool_val = expr pred_builder 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
   (* Implement for loops as while loops *)
   | SFor (e1, e2, e3, body) -> stmt builder
       ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )

in

 (* Build the code for each statement that exist outside of files *)
```

```
  let builder = stmt builder (SBlock statements) in

  (* Add a return for the simulated main function *)
  L.build_ret (L.const_int i32_t 0) builder
 in

  (List.iter build_function_body functions, build_statements statements);
  the_module  (* return the LLVM module result *)
```

7.11 compyled.ml

```
(* Top-level of the Jpie compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

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: ./jpie.native [-a|-s|-l|-c] [file.jpie]" 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 = Parser.program Scanner.tokenize 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)
```

---

8 tests

demo-triangle-test.cp

```
## Check if a triangle of positive area is possible with the given angles ##
## This example highlights "dangling else", as well as AND, OR, and comparison operators ##

## Adapted from: ##
##
https://www.geeksforgeeks.org/check-if-a-triangle-of-positive-area-is-possible-with-the-given-an
gles/ ##


def void isTriangleExists(int a, int b, int c)
{
   ## Checking if the sum of three ##
   ## angles is 180 and none of ##
   ## the angles is zero ##

   if(a != 0 and b != 0 and c != 0 and (a + b + c) == 180)
   {
      # Checking if sum of any two angles
      # is greater than equal to the third one

      if((a + b) >= c or (b + c)>= a or (a + c)>= b)
      {
         print("YES");
      }
      else
```

```
        {
            print("NO");
        }
    }
    else
    {
        print("NO");
    }
}


int a;
a = 50;
int b;
b = 60;
int c;
c = 70;

isTriangleExists(50, 60, 70);
```

---

fail-assign-array-index1.cp

```
array x [int 3];
x[1] = 2.2;
print(x[1]);
```

---

fail-assign1.cp

```
bool i;
i = 1;
print(i);
```

---

fail-assign2.cp

```
int i;
i = "hello";
print(i);
```

fail-assign3.cp

```
string s;
s = 1;
print(s);
```

fail-assign4.cp

```
int x;
x = 3 / 2.5;
```

fail-divide-int-float.cp

```
2 / 3.5;
```

fail-duplicate-func-name.cp

```
## ensure duplicate function names raise an error ##

def int add(int a, int b) {
  return a + b;
}

def int add(int a, int b) {
  return a + b;
}

add(2, 3);
add(3, 4);
```

fail-duplicate-var-name.cp

```
## ensure duplicate variable names raise an error ##

int x;
int x;
```

fail-expr1.cp

```
int i;

i = 1;

string s;
s = "hello";

i + s;
```

fail-expr2.cp

```
int i;

i = 1;

float s;
s = 3.14;

i + s;
```

fail-for1.cp

```
## test to make sure this fails on an undefined variable (j)##
int i;
for (i = 0; j < 10 ; i = i + 1) {
        print(j);
}
```

---

fail-for2.cp

```
## boolean expression not in the for loop ##
int i;
for (i = 0; i ; i = i + 1) {
        print(i);
}
```

---

fail-initialize-in-func.cp

```
## purpose of this test is to initialize a local var in a function ##

def int add_3_to(int a, int b) {
  int c = 3.2;
  return a + b + c;
}

int a;
a = add_3_to(39, 3);
print(a);
```

---

fail-initialize-var-in-func1.cp

```
## purpose of this test is to initialize a local string var in a function ##

def string gen_str(int a) {
  string c = "hello";
  return c;
}

string s;
## the below should not fail ##
print(c);
```

---

fail-initialize-var-to-func-return.cp

```
def int add(int a, int b) {
        return a + b;
}

int x = add(1, 2);
print(x);
```

---

fail-initialize-var-with-func.cp

```
## the purpose of this is to see if hoisting breaks ##

def int add(int a, int b) {
        return a + b + x;
}


int x = add(1, 2);
print(x);
```

---

fail-initialize1.cp

```
int x = 3.22;
print(x);
```

---

fail-initialize2.cp

```
string x = False;
print(x);
```

fail-return-no-func.cp

```
## ensure user cannot return from a statement outside of a function definition ##

int x;
x = 3;
return x;
```

fail-return-wrong-func-type.cp

```
## ensure functions fail if wrong return type declared ##

def string add(int a, int b) {
        return a + b;
}
```

fail-return1.cp

```
def int add (int a, int b) {
        return True;
}

int i;
int j;
i = 1;
```

```
j = 1;

add(i, j);
```

---

test-arithm-float-add.cp

```
print(3.3 + 4.3);
```

test-arithm-float-divide.cp

```
print(5.4 / 2.0);
```

test-arithm-float-multiply.cp

```
print(4.1 * 2.2);
```

test-arithm-float-sub.cp

```
print(4.3 - 1.1);
```

test-arithm-int-add.cp

```
print(40 + 9);
```

test-arithm-int-divide.cp

```
print(9 / 3);
```

test-arithm-int-multiply.cp

```
print(9 * 9);
```

test-arithm-int-sub.cp

```
print(40 - 9);
```

test-array-decl-in-func1.cp

```
## the purpose of this test is to ensure an array can be declared inside of a function ##
def int add(int a, int b) {
  array x [int 3];
  return a + b;
}

int a;
a = add(39, 3);
print(a);
```

test-array-dynamic-access.cp

```
array x [int 4];
x[0] = 2;
```

```
int i;
int c;
i = 0;
c = x[i];
print(c);
```

test-array-for-loop-assign.cp

```
array x [int 4];
int i;
int c;
for (i = 0 ; i < 4 ; i = i + 1) {
        x[i] = i;
        c = x[i];
        print(c);
}
```

test-array-for-loop.cp

```
array x [int 4];
x[0] = 2;
x[1] = 4;
x[2] = 6;
x[3] = 8;
int i;
int c;
for (i = 0 ; i < 4 ; i = i + 1) {
        c = x[i];
        print(c);
}
```

test-array-index-assign1.cp

```
array x [int 3];
x[1] = 2;
print(x[1]);
```

test-array-index-assign2.cp

```
array x [float 3];
x[1] = 2.3;
float a;
a = x[1];
print(a);
```

test-array-index-assign3.cp

```
 array x [string 3];
x[1] = "hello";
string s;
s = x[1];
print(s);
```

test-assign-arithm-add.cp

```
int x;
x = 3 + 3;
```

```
print(x);
```

test-assign-var-to-index-access.cp

```
def string gen_str(int a) {
  string c = "hello";
  return c;
}

string s;
print(c);
```

test-assign-var-to-var.cp

```
 int x;
x = 3;
int y;
y = x;
print(y);
```

test-assign1.cp

```
int x;
x = 1;
print(x);
```

test-assign2.cp

```
string x;
x = "hello";
print(x);
```

test-decl-array1.cp

```
array x [int 3];
```

test-decl-string1.cp

```
int x;
```

test-decl1.cp

```
int x;
```

test-elif1.cp

```
int x;
x = 1;
if (x > 1) {
        print("went to if branch");
}
else if (x == 1) {
        print("went to elif branch");
} else {
        print("went to else branch");
}
```

test-elif2.cp

```
int x;
x = 2;
if (x > 10) {
        print("went to if branch");
}
else if (x == 1) {
        print("went to elif branch");
}
else if (x == 2) {
	print("went to second elif branch");
} else {
	print("went to else branch");
}
```

test-elif3.cp

```
## test that the if, else if, else can go to a sub-branch ##

int x;
int y;
y = 3;
x = 1;
if (x > 10) {
	print("went to if branch");
}
else if (x == 1) {

	if (y == 3) {
		print("went to the inner if in the elif branch");
	}
}
else if (x == 2) {
	print("went to second elif branch");
} else {
	print("went to else branch");
}
```

test-elif4.cp

```
## test that the dangling else does not occur ##

int x;
int y;
y = 3;
x = 1;
```

```
if (x > 10) {
	print("went to if branch");
}
else if (x == 1) {

	if (y == 2) {
		print("went to the inner if in the elif branch");
	}
}
else if (x == 2) {
	print("went to second elif branch");
} else {
	print("went to else branch");
}
```

test-float1.cp

```
float x;
x = 4.56;
print(x);
```

test-for1.cp

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

test-func-with-main.cp

```
## test for ensuring the user can still declare some main function ##

def int add(int x, int y)
{
  return x + y;
}

def int main()
{
  int a;
  a = add(1, 3);

  print(a);
  return 0;
}

main();
```

test-func1.cp

```
def int add(int a, int b) {
  return a + b;
}

int a;
a = add(39, 3);
print(a);
```

test-func2.cp

```
## purpose of this test is to show functions can be declared after calls ##
int a;
a = add(39, 3);
print(a);

def int add(int a, int b) {
  return a + b;
}
```

test-func3.cp

```
## test where function is defined but not called ##
def int foo(int a) {
        return a;
}
```

test-func4.cp

```
## testing functions that have boolean values as arguments and if-else control flow ##

def void foo() {}

def int bar(int a, bool b, int c) {
        if (b) {
                return c - a;
        } else {
                return a + c;
        }
}

print(bar(8, False, 10));
```

test-function-call-nesting.cp

```
## to ensure functions call each other in the correct order ##

def void foo() {
        print("foo");

}

def void bar(int a, bool b, int c) {
        print("bar");
        foo();
}

def void say_hello() {
        print("Hello world");
        bar(1, False, 2);
}

say_hello();
```

test-gcd.cp

```
## find the GCD ##
## https://www.geeksforgeeks.org/gcd-in-python/ ##

## Python code to demonstrate naive ##
## method to compute gcd ( recursion ) ##

def int hcfnaive(int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return hcfnaive(b, a % b);
    }
}

print("The gcd of 60 and 48 is");
int result;
result = hcfnaive(60, 48);
print(result);
```

test-hello-world.cp

```
print("hello world");
```

test-hoisting1.cp

```
## to ensure that evaluating x works regardless of where it is declared/initialized ##

print(x);

int x = 1;
```

test-hoisting2.cp

```
## to ensure that evaluating x works regardless of where it is declared/initialized ##

print(x);

int x;
```

test-if-else-1

```
int x;
x = 1;
if (x == 1) {
	print("went to if branch");
} else {
	print("went to else branch");
}
```

test-if-else2.cp

```
int x;
x = 1;
if (x == 2) {
	print("went to if branch");
} else {
	print("went to else branch");
}
```

test-if-else3.cp

```
if (True) {
	print("went to if branch");
} else {
	print("went to else branch");
}
```

test-if-else4.cp

```
if (False) {
	print("went to if branch");
} else {
	print("went to else branch");
}
```

test-if-no-else1.cp

```
int x;
x = 1;
if (x == 1) {
	print("went to if branch");
}
```

test-initialize-var-in-func1.cp

```
## purpose of this test is to initialize a local var in a function ##
```

```
def int add_3_to(int a, int b) {
  int c = 3;
  return a + b + c;
}

int a;
a = add_3_to(39, 3);
print(a);
```

test-initialize-var-in-func2.cp

```
## purpose of this test is to initialize a local var in a function ##

def int add_3_to(int a, int b) {
  int c = 10;
  return a + b + c;
}

int c;
c = add_3_to(39, 3);
print(c);
```

test-initialize-var-in-func3.cp

```
## purpose of this test is to initialize a local string var in a function ##

def string gen_str(int a) {
  string c = "hello";
  return c;
}

string s;
s = gen_str(3);  ## not relevant. just needed until void works right ##
print(s);
```

test-initialize-var.cp

```
int x = 3;
print(x);
```

test-initialize-var2

```
string x = "hello";
print(x);
```

test-mod1.cp

```
print(3 % 3);
```

test-mod2.cp

```
print(3 % 4);
```

test-mod3.cp

```
print(4 % 3);
```

test-not1.cp

```
## test not unary operator ##
```

```
bool x = True;

if (not x) {
        print("should not go here");
} else {
        print("went to the correct branch");
}
```

test-not2.cp

```
## test not unary operator ##
bool x = False;

if (not x) {
        print("went to the correct branch");
} else {
        print("should not go here");
}
```

test-pass-func-as-arg.cp

```
## ensure that a function can be passed as an argument ##

def int add(int a, int b) {
  return a + b;
}

int a;
a = add(5, add(39, 3));
print(a);
```

test-reverse-function-call-nesting.cp

```
## show that functions can be defined in any order ##
```

```
def void say_hello() {
        print("Hello world");
        bar(1, False, 2);
}

def void bar(int a, bool b, int c) {
        print("bar");
        foo();
}

def void foo() {
        print("foo");

}

say_hello();
```

test-string-concat.cp

```
## testing string concatenation ##

string s1;
s1 = "hello ";

string s2;
s2 = " world!";

string s3;
s3 = s1 + s2;

print(s3);
```

test-string-mutation.cp

```
string s = "hello";
```

```
s = "hi";
print(s);
```

test-var-decl-in-func.cp

```
## ensure integer variables can be declared inside of a function ##

def int bar(int a, int b) {
	int x;
	x = 1;
	return b - x;
}

print(bar(1, 2));
```

test-var-decl-in-func2.cp

```
## ensure string variables can be declared inside of a function ##

def string bar(int a, int b) {
	string s;
	s = "I was declared and assigned in the functon";
	return s;
}

string a;
a = bar(1, 2);
print(a);
```

test-while1.cp

```
int i;
i = 0;
```

```
while (i < 5) {
        print(i);
        i = i + 1;
}
```

9 demo

test-demo-triangle.cp

```
## Check if a triangle of positive area is possible with the given angles ##
## This example highlights "dangling else", as well as AND, OR, and comparison operators
##

## Adapted from: ##
##
https://www.geeksforgeeks.org/check-if-a-triangle-of-positive-area-is-possible-with-the-give
n-angles/ ##


def void isTriangleExists(int a, int b, int c)
{
   ## Checking if the sum of three ##
   ## angles is 180 and none of ##
   ## the angles is zero ##

   if(a != 0 and b != 0 and c != 0 and (a + b + c) == 180)
   {
      ## Checking if sum of any two angles ##
      ## is greater than equal to the third one ##

      if((a + b) >= c or (b + c) >= a or (a + c) >= b)
      {
         print("YES");
      }
      else
      {
         print("NO");
      }
   }
   else
```

```
    {
        print("NO");
    }
}


int a;
a = 50;
int b;
b = 60;
int c;
c = 70;

isTriangleExists(50, 60, 70);
```

test-demo-triangle.cp

```
def in fib(int n) {
    if (n < 2) {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```