ProCSV

TAHIYA CHOWDHURY (TC2672) - LANGUAGE GURU, TESTER
TABARA NOSIBA (TN2341) - PROJECT MANAGER, TESTER
TAHSINA SAOSUN (TS2931) - SYSTEM ARCHITECT, TESTER

DECEMBER 19TH, 2018

Contents

1 Introduction

1.1 What is the ProCSV Language?

Data is an extremely important aspect of problem solving and for keeping inventory of important information in businesses, institutions, and other projects. Many developers use data in various formats on a daily basis to draw conclusions, support or disprove theories, and suggest solutions. One of the more ubiquitous formats of presenting data is in the CSV format. The ProCSV language is aimed at making it easier for programmers to process and work with CSV files. The language employs functions that allows users to manipulate CSV files in a few different ways that align with common tasks they might need to complete.

The ProCSV language solves issues pertaining to the manipulation of one or more CSV files. People who regularly analyze, manipulate, and compare data across multiple CSV files for data collection purposes may find it tedious and inefficient in most programming languages. ProCSV provides users with built-in functions that make tasks such as cleaning data, comparing, and parsing data stored within multiple CSV files easier. Anyone working with large CSV files will find this language helpful.

Our language is meant to streamline the parsing through CSV files. Since many institutions' data analysis process requires work to parse and visualize insights from traditionally formatted data collection formats, such as CSV. Simplifying this process would improve developers' productivity significantly and would also save companies millions of dollars in the process.

2 Usage

This section will briefly walk you through the steps that need to be followed in order to write, compile, and run your first program in ProCSV.

2.1 Getting Started

First, download the language project folder onto your local machine. Open up your terminal and change into the downloads directory, and then change into the `procsv` subdirectory. Now you are all ready to start running your first ProCSV programs!

2.2 Writing Your Program

Writing programs in our language is very similar to writing a program in C or Python. Create a new file using any text editor, with the extension `.pc` and start creating your program. Here is a simple "Hello World" program to help you get started:

```
// helloworld.pc
int main() {
    print_string("Hello world!");
    return 0;
}
```

2.3 Compiling the ProCSV Language

Make sure you are in the `procsvfinal` directory. Type `make` in your terminal, which will generate our ProCSV compiler. You can then check to make sure everything is running properly by running ./testall.sh, which should report if all the pass and fail test cases are running properly or not.

```
$ make
$ ./testall.sh
```

2.3.1 Compiling a .pc file

Navigate to the `procsvfinal` directory. Type `make` in your terminal, which will generate our ProCSV compiler. You can then compile the .pc file by following the commands below.

```
$ make
$ ./procsv.native < *name-of-file.pc* > temp.bc
$ clang -o temp.exe temp.bc printbig.o
$ ./temp.exe
```

3 Language Tutorial

After learning how to write and run a simple "Hello World" ProCSV program, it's time to get started on writing some more detailed programs. The ProCSV language allows for various data types, functions, and other such implementations. The following subsections will detail how you may use them and the types of programs you can write.

3.1 Integer

Various operations can be performed on integers in ProCSV, such as basic arithmetic operations or within user-defined mathematical functions. Below is an example of simple addition between integers:

```
int x = 10 + 25;
```

3.2 Floating Point Number

Floating point numbers, otherwise known as floats, are another form of numeric in ProCSV that has a fractional value. As with integers, operations can be performed on floats in ProCSV, such as basic arithmetic operations or within user-defined mathematical functions. Below is an example of simple addition between floating point numbers:

```
float x = 10.0 + 25.3;
```

3.3 String

Strings are a series of characters that may be assigned to a variable, printed to the standard output, or be passed as function arguments when needed. The following is an example of an assigned string in ProCSV:

```
string name = "Programming Languages and Translators";
```

3.4 Variable Declaration

Variables are symbols that are binded to a value by the user. However, each variable must be categorized as a certain type with its preceding term. In the following example, a user defines the variable with the name 'x', which is of type int (an integer) to the value 5. Variable assignment is done using the = operator which represents assignment.

```
int x = 5;
```

LOCAL VARIABLES

ProCSV utilizes local variables in the language. Local variables are variables in the language that are defined within specific functions and are only accessible within that given function.

GLOBAL VARIABLES

ProCSV allows for global variables in the language. Global variables are variables in the language that are defined in the beginning of the program and are accessible throughout the run of the program.

3.5 For Loop

For loops are a form of control flow in ProCSV that allows for users to iterate multiple times without manually writing statements that are repetitive. The following is an example of a for loop that prints "Hello World" 5 times:

```
for(int i = 0; i < 5; i++){
   print_string("Hello World");
}
```

3.6 While Loop

While loops are another form of control flow in ProCSV that allows for users to iterate multiple times without manually writing statements that are repetitive while a certain condition is true. The following is an example of a while loop that prints "Hello World" 5 times while a counter keeps track of the number of times it's iterating:

```
int i = 0;
while(i < 5){
   print_string("Hello World");
    i++;
}
```

3.7 If/else Conditions

If/else statements are the third form of control flow in ProCSV that allows for certain statements to be executed only if certain conditions are met. The following is an example of an if/else condition:

```
int i = 0;
if(i = 0){
   print_string("The variable i is set equal to 0.")
}
else{
   print_string("The variable i is set equal to something else.")
}
```

3.8 Library Functions

The ProCSV language includes many built-in library functions. These functions include *sim*(), *read_csv*(), *parse*(), and *find*(). Here is an example implementation of the find() function.

```
/*Takes in a CSV, looks for a specific keyword the user
is looking for & returns the relevant data with that keyword*/
char* roster_csv;
roster_csv = find("class_roster.csv", "Tabara");
print_string(roster_csv);
```

The following is an example of the parse() function implementation.

```
/*Takes in a CSV format of students and
returns a char* of the student names.*/
char* student_list;
student_list = parse("class_roster.csv", ",");
print_string(student_list);
```

Below is a snippet of the sim() function at work.

```
/*This function takes in two csv files
and returns the data that is the same */
string sameData;
sameData = sim("studentInfo.csv", "studentInfo2.csv");
print_string(sameData);
```

Below is a snippet of the *read_csv*() function.

```
/*This function takes in two csv files
and returns the data that is the same */
string sameData;
sameData = sim("studentInfo.csv", "studentInfo2.csv");
print_string(sameData);
```

4 Language Reference Manual

4.1 Lexical Elements

IDENTIFIERS
Identifier refer to the data types. They consist of one or more REs, lowercase letters, numbers, and underscores.

RESERVED KEYWORDS

int	boolean	float	string
void	print	while	for
return	if	else	merge
print_float	print_string	read_csv	sim
find	parse		

LITERALS
Integer literal: integers with one or more digits that may range anywhere between 0-9
Float literal: numbers with a fractional value; moreover, they contain a decimal (e.g. The integer 1 as a float would be 1.0)
Boolean literal: a value of either 'true' or 'false'
String literal: a series of one or more characters (e.g. "hello world" is a string literal that consists of the letters as characters as well as a space character)

OPERATORS

+, -, *, /, %	arithmetic integer operators
==, <, >, <=, >=	numerical integer operators
||, &&, !	logical operators
=	assignment

DELIMITERS
Parentheses: Parentheses are used to contain arguments to be used in function calls
Semicolon: Semicolons are used to signify the end of a programmed statement
Curly braces: used to contain the scope of a function or control flow
Commas: used to separate arguments or parameters in function calls, declarations, and data in the CSV files

WHITESPACE
Whitespace is only used to separate tokens.

Comments
Single line comments are denoted by /* */ and multiple line comments are also denoted by /* and */

```
/*This is a single line comment.*/
```

```
/*This is a multiple line comment.
ProCSV rocks.*/
```

4.2 Data Types

Primitive Data Types
Integers: we declare integers as type `int`.
For example:

```
int x = 10;
```

Floats: we declare floating point numbers as type `float`.
For example:

```
float x = 1.0;
```

Booleans: boolean values will be declared as type `bool` and can be either true or false. Example:

```
bool match = true;
```

Non-Primitive Data Types
Strings: strings are defined by the type `string`. They will be defined with two double quotes as shown, ‘‘ ’’. Example:

```
string intro = "Hello World";
```

Functions
Built-in Functions: functions predefined in the compiler. In PC, these are:
`print()` prints to the screen any data type passed as an argument
`print_string()` prints to the screen an argument of type `string`
`read_csv()` takes in a csv file and outputs the contents of the csv file
`parse()` takes in a csv file and outputs the contents of the file based on user-specified delimiter
`sim()` takes in two separate csv files, runs through each, compares the data of each, and outputs the same data among the two
`find()` takes in a csv file and a user specified keyword, runs through the csv file to find data with only that keyword, and outputs only the data that matches the keyword

Main Function: any code for .pc files will need to be executed within a main function as shown below

```
return_type main(){
    ...
    return return_type;
}
```

Control Flow Statements

Conditionals:

```
if (<bool>) {
    <expr>
}
else {
    <expr>
}
```

Loops:

```
for (int i=0; i<5; i=i+1) {
    <expr>
}
while (<bool>) {
    <expr>
}
```

4.3 Program Structure and Scoping Rules

4.3.1 Program Structure

All ProCSV programs need to be written and saved inside a single file with .pc file extension.

4.3.2 Scoping Rules

proCSV is a statically scoped language. As such, a variable always refers to its top level environment and an object is only visible to functions and operations that are written after it. For example, if x is a variable whose value is 10, another variable y = x + 2 cannot declared unless x comes before this declaration in the code. proCSV supports declaration of both global and local variables. Functions can only be defined in a global context. Our language does not support nested function declaration (declaration of a function within another function).

4.3.3 Global

Global variables can be declared outside of a function. These variables are visible throughout the entire program and can be accessed at any point of the code.

4.3.4 Local

Local variables have limited scope and are only visible within the function/ block of code where they are initialized.

4.4 Grammar

```
program -> decls EOF { $1 }

decls ->
   /* nothing */
 | vdecl
 | fdecl

typ ->
    INT | BOOL | VOID| STRING | FLOAT

stmt ->
    expr SEMI
  | RETURN SEMI
  | RETURN expr SEMI
  | LBRACE stmt_list RBRACE
  | IF LPAREN expr RPAREN stmt %prec NOELSE
  | IF LPAREN expr RPAREN stmt ELSE stmt
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
  | WHILE LPAREN expr RPAREN stmt
```

```
stmt_list ->
    /* nothing */
  | stmt_list stmt

expr ->
    LITERAL
  | TRUE
  | FALSE
  | ID
  | id INCREMENT
  | id DECREMENT
  | STRING_LIT
  | FLOAT_LIT
  | expr PLUS  expr
  | expr MINUS expr
  | expr TIMES expr
  | expr DIVIDE expr
  | expr MOD expr
  | expr EQ    expr
  | expr NEQ   expr
  | expr LT    expr
  | expr LEQ   expr
  | expr GT    expr
  | expr GEQ   expr
  | expr AND   expr
  | expr OR    expr
  | MINUS expr %prec NEG
  | NOT expr
  | expr ASSIGN expr
  | ID LPAREN actuals_opt RPAREN
  | LPAREN expr RPAREN

 decls ->
   /* nothing */
 | decls vdecl
 | decls fdecl

 vdecl ->
   typ ID SEMI

 fdecl ->
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE

 formals_opt ->
    /* nothing */
  | formal_list

 formal_list ->
    typ ID
  | formal_list COMMA typ ID

 actuals_opt ->
    /* nothing */
  | actuals_list

 actuals_list ->
    expr
  | actuals_list COMMA expr
```

5 Project Plan

5.1 Process

Before we had access to the microc suite, we first started the project by referencing older projects' codebases and attempting to understand the structure of the programs. We started our own repository from scratch. So, we programmed the simple scanner and parser first before attempting the more complicated portions. But, since we were running into issues through this process, once we had access to the microc, we started over by implementing a simple mod operator completely.

After making sure we understood the vertical slice of changes made, we edited the project to return an output of "hello world". Once that milestone was reached, we began building off of this compiler to implement the operators and functions we needed. Since we are a relatively small team of three people, we ended working on the language in pair programming style the most often. While we kept our role designations at the beginning of the semester in mind, we helped one another throughout the process frequently.

5.2 Project Timeline

1. First, we started the project by referencing to older projects that implemented languages similar to which wanted to accomplish with ours.
2. Once we had access to the microc, we switched gears and started to understand the codebase that made up microc.
3. We implemented the functioning mod operator all across the project, which allowed us to be able to understand specifics of the microc boilerplate code more closely.
4. The test suite was up and running to make sure we knew that the mod functionality was working for us, along with the rest of the project's code.
5. With the knowledge of changes necessary to make the mod operator working, we implemented $print\_string()$ and were able to print out "hello world" in our language.
6. We implemented the $parse()$, $read\_csv()$ functions along with the test cases for these functions.
7. The $find()$ and $sim()$ functions were implemented, along with the test cases for these.
8. Codes for the demo were written out.

5.3 Team and Roles

1. Tahiya Chowdhury - Language Guru, Tester
2. Tabara Nosiba - Project Manager, Tester
3. Tahsina Saosun - System Architect, Tester

The roles listed above are assigned roles that we agreed upon at the beginning of the project. Because our team consisted of only three members, each one of us had to take on the role of a tester along with our specific role. Although we started out with each of us focusing on our assigned role, as we continued to develop our language, we had to take on responsibilities outside of our roles. On many occasions, we opted to pair programming as we found it be more productive than programming individually.

5.4 Programming Style Guide

We all used the Sublime text editor, which made all of our files readily available for us to access. Features such as automatic indentation and function completion created ease in programming.

We utilized Bitbucket's git version control system for this project. We created a BitBucket repository for our project, and each pulled and committed the code we worked on separately. In addition, we used Bitbucket's built-in integration with Trello to keep track of the work that was completed over the course of the semester. Tasks were delegated to each members and larger tasks included checklists, which were used to keep track of what needed to be done at a time.
In order to maintain a clean master branch, we created another branch called 'dev' to which we merged our individual branch work so that we never pushed code that didn't work. We set up our master so that it was protected from the dev branch and required pull requests. The final working code was pushed to the master branch at the end of our completion of the project.

One very useful technique that our team was able to pick up halfway through our process was to work in vertical slices on features that we wanted to implement into our language. For instance, we began with the mod (otherwise known as module) operator which we decided to implement starting with the scanner. After including the necessary token, we moved on to the parser and the abstract syntax tree files in order to ensure that our compiler can parse and recognize the mod operator when used in an expression or statement. Next, we implemented all the necessary corresponding changes in semant and codegen. We then, finally, generated tests cases where the operator would be used to ensure that our language could fully accept statements or conditions that used the operator and resulted in the correct output.

As for languages, we used OCaml LLVM for the abstract syntax tree builder, the LLVM IR, the semantic checker, and the code generation. We used OCamllex for our scanner and ocamlyacc for the parser. We used C to implement all of our built in functions.

Tab Size: 2

Commenting Convention:

Multi-line line comments:

```
(* commented line 1

   commented line 2

   commented line 3 *)
```

or

```
/* commented line 1

   commented line 2

   commented line 3 */
```

5.4.1 Guidelines

We have adhered to the following rules throughout our development process:

1. Always indent appropriately to make code more readable

2. Name variables following the camel case convention

3. Always create a seperate branch for your code and merge only after all test cases pass

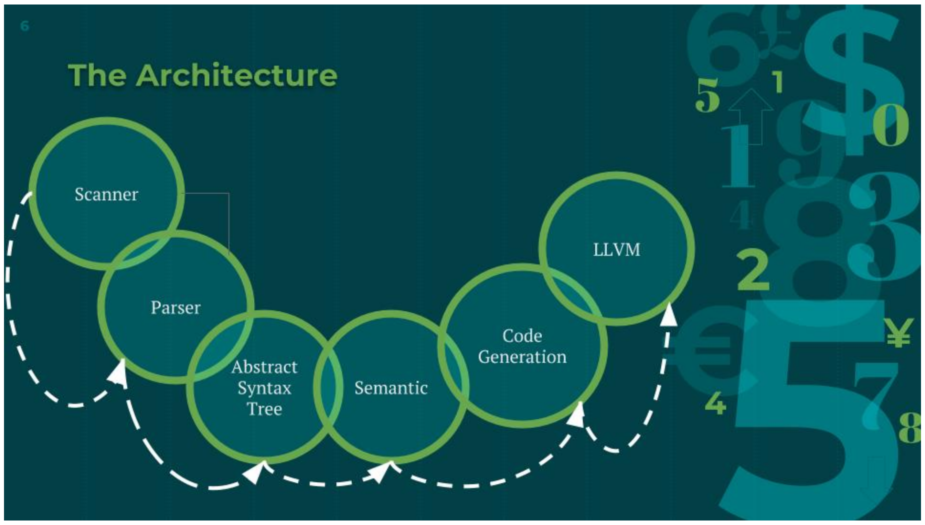
6. Once you start implementing a new feature, only move on to the next feature after writing all the codes necessary to make that feature work (starting from scanner to the test suite)

5. Come up with function names that tell the reader what the function does

6. Resolve all warnings generated before pushing code to the dev branch

6 Architectural Design

The compiler begins with the source code, passes that through the scanner and tokenizes it. This output is then passed through a parser, from which an abstract syntax tree is constructed. Then, the semantic analyzer checks the semantics of the program to detect any issues in structures, declarations, arguments, etc., and then passes that output through the code generator. Finally, this output is translated into LLVM code.



7 Test Plan

Below we have included a few programs written in proCSV that demonstrate most of the functionalities of our language.

7.1 Finding potential TAs

The following program demonstrates an example of how a professor can find information regarding students who can potentially TA for their class by checking whether or not they have taken the course and its prerequisite and have earned a grade of A in both of the courses. Each one of the two input files are CSV files that contain a list of students who have taken two specific courses and their information.

```
// demo1.pc
int main(){
   string potential_TAs;

   print_string("Potential TAs: ");
   print_string("");

   /* finds and stores all the data corresponding to students who have
       taken both plt and cst (finds similar data)*/
   potential_TAs = sim("plt_class_roster.csv","cst_class_roster.csv");
   print_string(potential_TAs);
   print_string("");

    /* Finds data corresponding to students who have earned a grade of A
        in both of the courses*/
   print_string("Qualified TAs: ");
   find(potential_TAs, "A");

   return 0;
}
```

```
Output:

Potential TAs:
Name, UNI, School, Year, Grade
Tahsina, ts2931, BC, 2019, B
Rida, ra2005, CC, 2020, A
Sam, sd2345, SEAS, 2019, A
Callie, cg1010, GS, 2021, C

Qualified TAs: Match on line: 3

Rida, ra2005, CC, 2020, A
Match on line: 4
```

```
Sam, sd2345, SEAS, 2019, A
```

7.2 Parsing Data by Delimiter

The following program demonstrates an example of how a user can parse a CSV file by a specific delimiter. In this example, we choose to delimit by comma.

```
// demo2.pc

int main(){
  string data;
  string parsed_data;

  /* calls the built in function read_csv in order to read the .csv file
      */
  data = read_csv("test_read_csv.csv");

  /*parses the data by comma and stores in variable*/
  parsed_data = parse(data, ",");

  print_string(parsed_data);
  return 0;
}
```

```
Output:

Name
 Year
 School
 GPA
Ana
 2022
 SEAS
 4.0
Mona
 2020
 BC
 3.7
Sarah
 2018
 CC
 3.3
Holden
 2021
 SEAS
 2.7
```

7.3 Test Cases

Our project has numerous test cases since the beginning of our development process. These test cases were used to check whether our implementations of functions, operators, or types could be recognized and interpreted by the compiler. The following test cases represent important implementations in the ProCSV language that allow for multiple functionalities.

Integers

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

int main()
{
  print( add(17, 25) );
  return 0;
}
```

```
Output:
42
```

Floats

```
float foo(float x, float y){
    return x + y;
}

int main(){
     float f;
     f = foo(1.0, 1.0);
     print_float(f);
     return 0;
}
```

```
Output:
2.000000
```

Mod Operator

```
int main()
{
```

```
int a;
a = 42;
a = a % 3;
print(a);
return 0;
}
```

```
Output:
0
```

Operations

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

```
Output:
3
-1
2
50
99
```

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

Arithmetic Operation

```
int main()
{
  print(39 + 3);
  return 0;
}
```

```
Output:
42
```

Variable Declaration and Assignment

```
int main(){

    int i;
    i = 0;
    print(i);
}
```

```
Output:
0
```

Decrement

```
int main(){
    int i;
    i = 10;
    i--;
    print(i);
}
```

```
Output:
9
```

Increment

```
int main(){
int i;
i = 0;
i++;
print(i);
return 0;
}
```

```
Output:
1
```

Global Variables

```
int a;
int b;

void printa()
{
  print(a);
}

void printb()
{
  print(b);
}

void incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
```

```
{
  a = 42;
  b = 21;
  printa();
  printb();
  incab();
  printa();
  printb();
  return 0;
}
```

```
Output:
42
21
43
22
```

Local Variables

```
void foo(bool i)
{
  int i; /* Should hide the formal i */

  i = 42;
  print(i + i);
}

int main()
{
  foo(true);
  return 0;
}
```

```
Output:
84
```

Fibonacci

```
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

int main()
```

```
{
  print(fib(0));
  print(fib(1));
  print(fib(2));
  print(fib(3));
  print(fib(4));
  print(fib(5));
  return 0;
}
```

```
Output:
1
1
2
3
5
8
```

Great Common Denominator

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

int main()
{
  print(gcd(2,14));
  print(gcd(3,15));
  print(gcd(99,121));
  return 0;
}
```

```
Output:
2
3
11
```

If Statement

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

```
Output:
42
17
```

For Loop

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

```
Output:
0
1
2
3
4
42
```

While Loop

```
int main()
{
  int i;
  i = 5;
  while (i > 0) {
    print(i);
    i = i - 1;
  }
  print(42);
  return 0;
}
```

```
Output:
```

```
5
4
3
2
1
42
```

read_csv

```
int main(){
  string buffer;
  buffer = read_csv("test_read_csv.csv");
  print_string(buffer);
  return 0;
}
```

```
Output:
Name, Year, School, GPA
Mona, 2020, BC, 3.7
Sarah, 2018, CC, 3.3
Holden, 2021, SEAS, 2.7
```

parse

```
int main(){
  string buffer;
  string parsed_buffer;
  buffer = read_csv("test_read_csv.csv");
  parsed_buffer = parse(buffer, ",");
  print_string(parsed_buffer);
  return 0;
}
```

```
Name
 Year
 School
 GPA
Ana
 2022
 SEAS
 4.0
Mona
 2020
 BC
 3.7
```

```
Sarah
 2018
 CC
 3.3
Holden
 2021
 SEAS
 2.7
```

sim

```
int main(){
  string buffer;
  buffer = sim("test_read_csv.csv", "test_read_csv2.csv");
  print_string(buffer);
  return 0;
}
```

```
Output:

Name, Year, School, GPA
Ana, 2022, SEAS, 4.0
Mona, 2020, BC, 3.7
Sarah, 2018, CC, 3.3
Holden, 2021, SEAS, 2.7
```

find

```
int main(){
  string buffer;
  buffer = find("test_read_csv.csv", "BC");
  print_string(buffer);
  return 0;
}
```

```
Output:
Match on line: 3
Mona, 2020, BC, 3.7
```

7.4 Automation

We used the script called testall.sh to automatically run all of our tests. The script outputs either OK or FAILED indicating whether or not the test passed or failed.

The output when ./testall.sh runs is as follows:

```
test-add1...OK
test-arith1...OK
test-arith2...OK
test-arith3...OK
test-dec...OK
test-fib...OK
test-find...test-float...OK
test-for1...OK
test-for2...OK
test-func1...OK
test-func2...OK
test-func3...OK
test-func4...OK
test-func5...OK
test-func6...OK
test-func7...OK
test-func8...OK
test-gcd...OK
test-gcd2...OK
test-global1...OK
test-global2...OK
test-global3...OK
test-hello...OK
test-if1...OK
test-if2...OK
test-if3...OK
test-if4...OK
test-if5...OK
test-inc...OK
test-local1...OK
test-local2...OK
test-mod...OK
test-ops1...OK
test-ops2...OK
test-parse...OK
test-print_float...OK
test-print_float...OK
test-print_string...OK
test-printbig...OK
test-read_csv...OK
test-sim...OK
test-var1...OK
test-var2...OK
test-vardec1...OK
test-vardec2...OK
test-while1...OK
test-while2...OK
```

```
fail-add...OK
fail-assign1...OK
fail-assign2...OK
fail-assign3...OK
fail-dead1...OK
fail-dead2...OK
fail-expr1...OK
fail-expr2...OK
fail-float...OK
fail-for...OK
fail-for1...OK
fail-for2...OK
fail-for3...OK
fail-for4...OK
fail-for5...OK
fail-func1...OK
fail-func2...OK
fail-func3...OK
fail-func4...OK
fail-func5...OK
fail-func6...OK
fail-func7...OK
fail-func8...OK
fail-func9...OK
fail-funcdec2...OK
fail-funcdec3...OK
fail-global1...OK
fail-global2...OK
fail-if...OK
fail-if1...OK
fail-if2...OK
fail-if3...OK
fail-increment...OK
fail-nomain...OK
fail-parse...OK
fail-return1...OK
fail-return2...OK
fail-sim...OK
fail-vardec1...OK
fail-while...OK
fail-while1...OK
fail-while2...OK
```

8 Lessons Learned

8.1 Tahiya

Working on this project has definitely helped me learn more about functional programming, time management, and how to work in a group. Not only was I able to learn how to design and implement a simple compiler, but I was also able to realize the importance of managing our time well, delegating tasks, keeping tabs on each other, and having a clean working environment. Picking up functional programming by coding in OCaml and leaning about how modern compilers are implemented has made me a better programmer overall and added one more skill to my toolkit as a developer. I was able to take a step back from looking at problems from an object-oriented perspective, and instead, look at them from a functional perspective. Learning how to work on features in vertical slices perhaps the most valuable skill I have mastered this semester. In addition, I have learned the importance of fully understanding a code base before attempting to make changes and learning when to ask for help.

8.2 Tabara

Although we had many roadblocks throughout the course of the semester that only allowed us to implement a very simple language, I learned and grew a lot through the semester by becoming aware of our own mistakes and gaining more knowledge about languages and compilers. I also learned about the importance of a strong work flow and time management that could have heavily influenced the outcomes of our project. While our idea was reasonable and had a strong motivation, we were not able to implement all aspects of our language proposed earlier due to falling behind in our implementations. I learned valuable programming work-flow techniques that would have saved time for our team earlier such as working in vertical slices on features. It is also very important to flesh out the details of the language with the final demo in mind so that features can be better prioritized based on usage.

8.3 Tahsina

This project was a huge learning curve for me. I think reflecting back on it allows me to understand some of the management and design choice mistakes we made. I learned that it is important to really start by working off of a boilerplate codebase that you actually understand. Vertical slices for features and learning to estimate tasks better were other important skills I learned. And it's also extremely important to ask for help more often. This project allowed me to get better with knowing when to ask for help and when to keep troubleshooting something.

9 Appendix

9.1 scanner.mll

```
(* Ocamllex scanner for ProCSV *)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"    { comment lexbuf }        (* Comments *)
| '('     { LPAREN }
| ')'     { RPAREN }
| '{'     { LBRACE }
| '}'     { RBRACE }
| ';'     { SEMI }
| ','     { COMMA }
| '+'     { PLUS }
| '-'     { MINUS }
| '*'     { TIMES }
| '/'     { DIVIDE }
| '='     { ASSIGN }
| "++"    { INCREMENT }
| "--"    { DECREMENT }
| "=="    { EQ }
| "!="    { NEQ }
| '<'     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| "&&"    { AND }
| "||"    { OR }
| "!"     { NOT }
| "%"     { MOD }
| "if"    { IF }
| "else"  { ELSE }
| "for"   { FOR }
| "while" { WHILE }
| "return" { RETURN }
| "int"   { INT }
|"float" { FLOAT }
| "bool"  { BOOL }
| "void"  { VOID }
| "true"  { TRUE }
| "false" { FALSE }
| "string" { STRING }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['0'-'9']+'.'['0'-'9']* | ['0'-'9']*'.'['0'-'9']+
   as lxm { FLOAT_LIT(float_of_string lxm)}
```

```
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"' (([^ '"'] | "\\\"")* as strlit )'"' { STRING_LIT(strlit)}
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

9.2 parser.mly

```
/* Ocamlyacc parser for MicroC */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT MOD DECREMENT INCREMENT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID STRING
%token <int> LITERAL
%token <float> FLOAT_LIT
%token <string> ID
%token <string> STRING_LIT

%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

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

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $1;
    fname = $2;
    formals = $4;
    locals = List.rev $7;
    body = List.rev $8 } }

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

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

typ:
    INT { Int }
  | BOOL { Bool }
  | VOID { Void }
  | STRING { String }
  | FLOAT {Float}

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

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

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

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

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

id:
  ID             { Id($1) }

expr:
    LITERAL        { Literal($1) }
  | TRUE           { BoolLit(true) }
  | FALSE          { BoolLit(false) }
  | ID             { Id($1) }
  | id INCREMENT { Pop($1, Inc) }
  | id DECREMENT { Pop($1, Dec) }
  | STRING_LIT     { StringLit($1) }
  | FLOAT_LIT      { FloatLit($1) }
  | expr PLUS  expr { Binop($1, Add, $3) }
  | expr MINUS expr { Binop($1, Sub, $3) }
  | expr TIMES expr { Binop($1, Mult, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr MOD expr { Binop($1, Mod, $3) }
  | expr EQ    expr { Binop($1, Equal, $3) }
  | expr NEQ   expr { Binop($1, Neq, $3) }
  | expr LT    expr { Binop($1, Less, $3) }
  | expr LEQ   expr { Binop($1, Leq, $3) }
  | expr GT    expr { Binop($1, Greater, $3) }
  | expr GEQ   expr { Binop($1, Geq, $3) }
  | expr AND   expr { Binop($1, And, $3) }
  | expr OR    expr { Binop($1, Or,  $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr       { Unop(Not, $2) }
  | expr ASSIGN expr { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list { List.rev $1 }

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

9.3 ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater |
    Geq |
         And | Or | Mod

type pop =
  | Dec
  | Inc

type uop = Neg | Not

type typ = Int | Bool | Void | String | Float

type bind = typ * string

type expr =
    Literal of int
  | BoolLit of bool
  | StringLit of string
  | FloatLit of float
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Pop of expr * pop
  | Assign of expr * 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 func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)

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

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

let string_of_pop = function
    Inc -> "++"
  | Dec -> "--"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | StringLit(s) -> s
  | FloatLit(f) -> string_of_float f
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Pop(v, p) -> string_of_expr v ^ string_of_pop p
  | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
  | 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_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Void -> "void"
  | String -> "string"
  | Float -> "float"

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

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

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

9.4 sement.ml

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

open Ast

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
   n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void exceptf = function
      (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in
```

```
(* Raise an exception of the given rvalue type cannot be assigned to
   the given lvalue type *)
let check_assign lvaluet rvaluet err =
  if(String.compare (string_of_typ lvaluet) (string_of_typ rvaluet) ==
      0)
  then lvaluet
  else raise err
  (*if lvaluet == rvaluet then lvaluet else raise err*)
in

(**** Checking Global Variables ****)

List.iter (check_not_void (fun n -> "illegal void global " ^ n))
    globals;

report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd
    globals);

(**** Checking Functions ****)

if List.mem "print" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function print may not be defined")) else ();

report_duplicate (fun n -> "duplicate function " ^ n)
  (List.map (fun fd -> fd.fname) functions);

(* Function declaration for a named function *)
let built_in_decls = StringMap.add "print"
   { typ = Void; fname = "print"; formals = [(Int, "x")];
     locals = []; body = [] } (StringMap.add "printb"
   { typ = Void; fname = "printb"; formals = [(Bool, "x")];
     locals = []; body = [] } (StringMap.add "print_string"
   { typ = Void; fname = "print_string"; formals = [(String, "x")];
     locals = []; body = [] } (StringMap.add "print_float"
   { typ = Void; fname = "print_float"; formals = [(Float, "x")];
     locals = []; body = [] } (StringMap.add "read_csv"
   { typ = String; fname = "read_csv"; formals = [(String, "x")];
      locals = []; body = [] } (StringMap.add "parse"
   { typ = String; fname = "parse"; formals = [(String, "x"); (String,
       "x")];
     locals = []; body = [] }(StringMap.add "sim"
   { typ = String; fname = "sim"; formals = [(String, "x"); (String,
       "x")];
   locals = []; body = [] }(StringMap.add "merge"
   { typ = String; fname = "merge"; formals = [(String, "x"); (String,
       "x"); (String, "x")];
     locals = []; body = [] } (StringMap.add "find"
   { typ = String; fname = "find"; formals = [(String, "x"); (String,
       "x")];
     locals = []; body = [] } (StringMap.singleton "printbig"
   { typ = Void; fname = "printbig"; formals = [(Int, "x")];
     locals = []; body = [] })))))))))
 in

let function_decls = List.fold_left (fun m fd -> StringMap.add
    fd.fname fd m)
                     built_in_decls functions
in

let function_decl s = try StringMap.find s function_decls
     with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

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

let check_function func =

  List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
    " in " ^ func.fname)) func.formals;

  report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^
      func.fname)
    (List.map snd func.formals);

  List.iter (check_not_void (fun n -> "illegal void local " ^ n ^
    " in " ^ func.fname)) func.locals;

  report_duplicate (fun n -> "duplicate local " ^ n ^ " in " ^
      func.fname)
    (List.map snd func.locals);

  (* Type of each variable (global, formal, or local *)
  let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t m)
    StringMap.empty (globals @ func.formals @ func.locals )
  in

  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in

  (* Return the type of an expression or throw an exception *)
let rec expr = function
   Literal _ -> Int
| BoolLit _ -> Bool
| StringLit _ -> String
| FloatLit _ -> Float
| Id s -> (type_of_identifier s)
| Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
```

```
    (match op with
    Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Int -> Int
   | Add | Sub | Mult | Div when t1 = Float && t2 = Float -> Float
    | Equal | Neq when t1 = t2 -> Bool
    | Less | Leq | Greater | Geq when t1 = Int && t2 = Int -> Bool
   | Less | Leq | Greater | Geq when t1 = Float && t2 = Float -> Bool
    | And | Or when t1 = Bool && t2 = Bool -> Bool
   | _ -> raise (Failure ("illegal binary operator " ^
           string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
           string_of_typ t2 ^ " in " ^ string_of_expr e))
    )
| Unop(op, e) as ex -> let t = expr e in
  (match op with
    Neg when t = Int -> Int
   | Neg when t = Float -> Float
    | Not when t = Bool -> Bool
   | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op
       ^
         string_of_typ t ^ " in " ^ string_of_expr ex)))

| Pop(e, op) as ex -> let t = expr e in
      (match op with
       | Inc | Dec -> (match t with
                        Int -> Int
                      | _ -> raise (Failure ("illegal postfix operator
                          " ^ string_of_pop op ^ " used with a " ^
                          string_of_typ t ^ " in " ^ string_of_expr
                          ex))))
| Noexpr -> Void

| Call(fname, actuals) as call -> let fd = function_decl fname in
       if List.length actuals != List.length fd.formals then
         raise (Failure ("expecting " ^ string_of_int
           (List.length fd.formals) ^ " arguments in " ^ string_of_expr
                call))
       else
         List.iter2 (fun (ft, _) e -> let et = expr e in
            ignore (check_assign ft et
              (Failure ("illegal actual argument found " ^
                   string_of_typ et ^
              " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr
                   e))))
           fd.formals actuals;
         fd.typ

| Assign(e1, e2) as ex ->
(match e1 with
  Id s ->
    let lt = type_of_identifier s and rt = expr e2 in
```

```
        check_assign (lt) (rt) (Failure ("illegal assignment " ^
            string_of_typ lt ^ " = " ^
                      string_of_typ rt ^ " in " ^ string_of_expr ex))
)
in

  let check_bool_expr e = if expr e != Bool
   then raise (Failure ("expected Boolean expression in " ^
       string_of_expr e))
   else () in

  (* Verify a statement or throw an exception *)
  let rec stmt = function
 Block sl -> let rec check_block = function
        [Return _ as s] -> stmt s
      | Return _ :: _ -> raise (Failure "nothing may follow a return")
      | Block sl :: ss -> check_block (sl @ ss)
      | s :: ss -> stmt s ; check_block ss
      | [] -> ()
     in check_block sl
    | Expr e -> ignore (expr e)
    | Return e -> let t = expr e in if t = func.typ then () else
       raise (Failure ("return gives " ^ string_of_typ t ^ " expected
           " ^
                    string_of_typ func.typ ^ " in " ^ string_of_expr
                        e))

    | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
    | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
                             ignore (expr e3); stmt st
    | While(p, s) -> check_bool_expr p; stmt s
  in

  stmt (Block func.body)

in
List.iter check_function functions
```

9.5 codegen.ml

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

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html
Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)

module L = Llvm
module A = Ast
module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "ProCSV"
  and i32_t = L.i32_type context
  and i8_t  = L.i8_type  context
  and i1_t  = L.i1_type  context
  and f_t   = L.double_type context
  and ptr_t = L.pointer_type (L.i8_type (context))
  and void_t = L.void_type context in

  let ltype_of_typ = function
      A.Int -> i32_t
    | A.Float -> f_t
    | A.Bool -> i1_t
    | A.String -> ptr_t
    | A.Void -> void_t in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m (t, n) =
      let init = L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in

  (* Declare printf(), which the print built-in function will call *)
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |]
      in
  let printf_func = L.declare_function "printf" printf_t the_module in

  (* Declare the built-in printbig() function *)
  let printbig_t = L.function_type i32_t [| i32_t |] in
  let printbig_func = L.declare_function "printbig" printbig_t
      the_module in

  (* Declare the built-in read_csv() function *)
  let read_csv_t = L.function_type ptr_t [| ptr_t |] in
  let read_csv_func = L.declare_function "read_csv" read_csv_t
      the_module in
  (* Declare the built-in sim() function *)
  let sim_t = L.function_type ptr_t [| ptr_t; ptr_t |] in
  let sim_func = L.declare_function "sim" sim_t the_module in

  (* Declare the built-in parse() function *)
  let parse_t = L.function_type ptr_t [| ptr_t; ptr_t |] in
  let parse_func = L.declare_function "parse" parse_t the_module in

  (* Declare the built-in merge() function *)
let merge_t = L.function_type ptr_t [| ptr_t; ptr_t; ptr_t |] in
  let merge_func = L.declare_function "merge" merge_t the_module in

  (* Declare the built-in find() function *)
  let find_t = L.function_type ptr_t [| ptr_t; ptr_t |] in
  let find_func = L.declare_function "find" find_t the_module in

  (* Declare the built-in printbig() function
  let print_string_t = L.function_type i8_t [| L.pointer_type i8_t |] in
  let print_string_func = L.declare_function "print_string"
      print_string_t the_module in*)

  (* Define each function (arguments and return type) so we can call it
      *)
  let function_decls =
    let function_decl m fdecl =
      let name = fdecl.A.fname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
             fdecl.A.formals)
      in let ftype = L.function_type (ltype_of_typ fdecl.A.typ)
           formal_types in
      StringMap.add name (L.define_function name ftype the_module,
           fdecl) m in
    List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.A.fname function_decls
        in
    let builder = L.builder_at_end context (L.entry_block the_function)
        in
    let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in
    let float_format_str = L.build_global_stringptr "%f\n" "fmt" builder
        in
    (*let string_format_str = L.build_global_stringptr "%s\n" "fmt"
        builder in*)

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

      let add_local m (t, n) =
       let local_var = L.build_alloca (ltype_of_typ t) n builder
       in StringMap.add n local_var m in

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

    (* Return the value for a variable or formal argument *)
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars
    in
(* Construct code for an expression; return its value *)
    let rec llvalue_expr_getter builder = function
          A.Id s -> lookup s
    |_ -> raise (Failure ("in llvalue_expr_getter but not a dotop!"))
and
  (* Return addr of lhs expr *)
  (*let addr_of_expr expr builder g_map l_map = match expr with
    A.Id(id) -> (lookup g_map l_map id)
   | _ -> raise (Failure("addr not found"))

  in *)
    (* Construct code for an expression; return its value *)
   expr builder = function
      A.Literal i -> L.const_int i32_t i
    | A.FloatLit f -> L.const_float f_t f
    | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
    | A.StringLit s -> L.build_global_stringptr s "str" builder
    | A.Noexpr -> L.const_int i32_t 0
    | A.Id s -> L.build_load (lookup s) s builder
    | A.Binop (e1, op, e2) ->
      let e1' = expr builder e1
      and e2' = expr builder e2 in
        if (L.type_of e1' = f_t || L.type_of e2' = f_t) then
           (match op with
            A.Add    -> L.build_fadd
          | A.Sub    -> L.build_fsub
          | A.Mult   -> L.build_fmul
          | A.Div    -> L.build_fdiv
          | A.Equal  -> L.build_fcmp L.Fcmp.Oeq
          | A.Neq    -> L.build_fcmp L.Fcmp.One
          | A.Less   -> L.build_fcmp L.Fcmp.Olt
          | A.Leq    -> L.build_fcmp L.Fcmp.Ole
          | A.Greater -> L.build_fcmp L.Fcmp.Ogt
          | A.Geq    -> L.build_fcmp L.Fcmp.Oge
          | _ -> raise (Failure ("operator not supported for operand"))
          ) e1' e2' "tmp" builder
      else
        (match op with
        | A.Add    -> L.build_add
        | A.Sub    -> L.build_sub
        | A.Mult   -> L.build_mul
       | A.Div    -> L.build_sdiv
       | A.Mod    -> L.build_srem
        | A.And    -> L.build_and
        | A.Or     -> L.build_or
        | A.Equal  -> L.build_icmp L.Icmp.Eq
        | A.Neq    -> L.build_icmp L.Icmp.Ne
        | A.Less   -> L.build_icmp L.Icmp.Slt
        | A.Leq    -> L.build_icmp L.Icmp.Sle
        | A.Greater -> L.build_icmp L.Icmp.Sgt
        | A.Geq    -> L.build_icmp L.Icmp.Sge
          ) e1' e2' "tmp" builder
  | A.Unop(op, e) ->
    let e' = expr builder e in
    (match op with
     A.Neg    -> L.build_neg
    | A.Not    -> L.build_not) e' "tmp" builder

  | A.Pop(e, op) ->
   let e' = expr builder e in
        (match op with
        | A.Inc -> ignore(expr builder (A.Assign(e, A.Binop(e, A.Add,
            A.Literal(1))))); e'
        | A.Dec -> ignore(expr builder (A.Assign(e, A.Binop(e, A.Sub,
            A.Literal(1))))); e')


  (*|  A.Assign (e1, e2) -> let l_val = (addr_of_expr e1 builder g_map
       l_map) in
    let e2' = expr builder g_map l_map e2 in
    ignore (L.build_store e2' l_val builder); e2' *)
      | A.Assign (lhs, e2) -> let e2' = expr builder e2 in
      (match lhs with
      A.Id s ->ignore (L.build_store e2' (lookup s) builder); e2'
       |_ -> raise (Failure("nooo"))
          )
(*      |_ -> raise (Failure("can't match in assign"))
  *)
```

```
   | A.Call ("print", [e]) | A.Call ("printb", [e])->
  L.build_call printf_func [| int_format_str ; (expr builder e) |]
    "printf" builder
   | A.Call ("print_string", [e]) ->
   L.build_call printf_func [|(expr builder e) |] "print_string"
       builder
   | A.Call ("print_float", [e]) ->
   L.build_call printf_func [|float_format_str;(expr builder e)|]
       "print_float" builder
   | A.Call ("printbig", [e]) ->
  L.build_call printbig_func [| (expr builder e) |] "printbig" builder
   | A.Call ("read_csv", [e]) ->
       L.build_call read_csv_func [| (expr builder e) |] "read_csv"
           builder
   (* | A.Call ("sim", args) ->
       L.build_call sim_func [|(expr builder e1) , (expr builder e2)|]
           "sim" builder *)
   | A.Call ("sim", act) ->
     let actuals = List.rev (List.map (expr builder) (List.rev act))
         in
      L.build_call sim_func (Array.of_list actuals) "sim" builder
   | A.Call ("parse", act) ->
     let actuals = List.rev (List.map (expr builder) (List.rev act))
         in
      L.build_call parse_func (Array.of_list actuals) "parse" builder
   | A.Call ("find", act) ->
     let actuals = List.rev (List.map (expr builder) (List.rev act))
         in
      L.build_call find_func (Array.of_list actuals) "find" builder
   | A.Call ("merge", act) ->
     let actuals = List.rev (List.map (expr builder) (List.rev act))
         in
      L.build_call merge_func (Array.of_list actuals) "merge" builder
   | A.Call (f, act) ->
      let (fdef, fdecl) = StringMap.find f function_decls in
 let actuals = List.rev (List.map (expr builder) (List.rev act)) in
 let result = (match fdecl.A.typ with A.Void -> ""
                                    | _ -> f ^ "_result") in
      L.build_call fdef (Array.of_list actuals) result builder
 in

 (* Invoke "f builder" if the current block doesn't already
    have a terminal (e.g., a branch). *)
 let add_terminal builder f =
   match L.block_terminator (L.insertion_block builder) with
Some _ -> ()
   | None -> ignore (f builder) in

 (* Build the code for the given statement; return the builder for
    the statement's successor *)
  let rec stmt builder = function
 A.Block sl -> List.fold_left stmt builder sl
    | A.Expr e -> ignore (expr builder e); builder
    | A.Return e -> ignore (match fdecl.A.typ with
   A.Void -> L.build_ret_void builder
 | _ -> L.build_ret (expr builder e) builder); builder
    | A.If (predicate, then_stmt, else_stmt) ->
       let bool_val = expr builder predicate in
  let merge_bb = L.append_block context "merge" the_function in

  let then_bb = L.append_block context "then" the_function in
  add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
    (L.build_br merge_bb);

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

  ignore (L.build_cond_br bool_val then_bb else_bb builder);
  L.builder_at_end context merge_bb
    | A.While (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

    | A.For (e1, e2, e3, body) -> stmt builder
     ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ]
         )
  in

  (* Build the code for each statement in the function *)
  let builder = stmt builder (A.Block fdecl.A.body) in

  (* Add a return if the last block falls off the end *)
  add_terminal builder (match fdecl.A.typ with
      A.Void -> L.build_ret_void
    | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
in

List.iter build_function_body functions;
```

```
  the_module
```

9.6 procsv.ml

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

module StringMap = Map.Make(String)

type action = Ast | 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 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: ./procsv.native [-a|-l|-c] [file.pc]" 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.token lexbuf in
  Semant.check ast;
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
      ast))
  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

9.7 printbig.c

```
/*
 *  A function illustrating how to link C code to code generated from
     LLVM
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*
```

```
 * Font information: one byte per row, 8 rows per character
 * In order, space, 0-9, A-Z
 */
static const char font[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x1c, 0x3e, 0x61, 0x41, 0x43, 0x3e, 0x1c, 0x00,
  0x00, 0x40, 0x42, 0x7f, 0x7f, 0x40, 0x40, 0x00,
  0x62, 0x73, 0x79, 0x59, 0x5d, 0x4f, 0x46, 0x00,
  0x20, 0x61, 0x49, 0x4d, 0x4f, 0x7b, 0x31, 0x00,
  0x18, 0x1c, 0x16, 0x13, 0x7f, 0x7f, 0x10, 0x00,
  0x27, 0x67, 0x45, 0x45, 0x45, 0x7d, 0x38, 0x00,
  0x3c, 0x7e, 0x4b, 0x49, 0x49, 0x79, 0x30, 0x00,
  0x03, 0x03, 0x71, 0x79, 0x0d, 0x07, 0x03, 0x00,
  0x36, 0x4f, 0x4d, 0x59, 0x59, 0x76, 0x30, 0x00,
  0x06, 0x4f, 0x49, 0x49, 0x69, 0x3f, 0x1e, 0x00,
  0x7c, 0x7e, 0x13, 0x11, 0x13, 0x7e, 0x7c, 0x00,
  0x7f, 0x7f, 0x49, 0x49, 0x49, 0x7f, 0x36, 0x00,
  0x1c, 0x3e, 0x63, 0x41, 0x41, 0x63, 0x22, 0x00,
  0x7f, 0x7f, 0x41, 0x41, 0x63, 0x3e, 0x1c, 0x00,
  0x00, 0x7f, 0x7f, 0x49, 0x49, 0x49, 0x41, 0x00,
  0x7f, 0x7f, 0x09, 0x09, 0x09, 0x09, 0x01, 0x00,
  0x1c, 0x3e, 0x63, 0x41, 0x49, 0x79, 0x79, 0x00,
  0x7f, 0x7f, 0x08, 0x08, 0x08, 0x7f, 0x7f, 0x00,
  0x00, 0x41, 0x41, 0x7f, 0x7f, 0x41, 0x41, 0x00,
  0x20, 0x60, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
  0x7f, 0x7f, 0x18, 0x3c, 0x76, 0x63, 0x41, 0x00,
  0x00, 0x7f, 0x7f, 0x40, 0x40, 0x40, 0x40, 0x00,
  0x7f, 0x7f, 0x0e, 0x1c, 0x0e, 0x7f, 0x7f, 0x00,
  0x7f, 0x7f, 0x0e, 0x1c, 0x38, 0x7f, 0x7f, 0x00,
  0x3e, 0x7f, 0x41, 0x41, 0x41, 0x7f, 0x3e, 0x00,
  0x7f, 0x7f, 0x11, 0x11, 0x11, 0x1f, 0x0e, 0x00,
  0x3e, 0x7f, 0x41, 0x51, 0x71, 0x3f, 0x5e, 0x00,
  0x7f, 0x7f, 0x11, 0x31, 0x79, 0x6f, 0x4e, 0x00,
  0x26, 0x6f, 0x49, 0x49, 0x4b, 0x7a, 0x30, 0x00,
  0x00, 0x01, 0x01, 0x7f, 0x7f, 0x01, 0x01, 0x00,
  0x3f, 0x7f, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
  0x0f, 0x1f, 0x38, 0x70, 0x38, 0x1f, 0x0f, 0x00,
  0x1f, 0x7f, 0x38, 0x1c, 0x38, 0x7f, 0x1f, 0x00,
  0x63, 0x77, 0x3e, 0x1c, 0x3e, 0x77, 0x63, 0x00,
  0x00, 0x03, 0x0f, 0x78, 0x78, 0x0f, 0x03, 0x00,
  0x61, 0x71, 0x79, 0x5d, 0x4f, 0x47, 0x43, 0x00
};

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

char* read_csv(char* filename){
  FILE* file = fopen(filename, "r");
  int file_length;
  int res;
  char* buffer;
  if (file == NULL){
    exit(1);
  }
  fseek(file, 0, SEEK_END);
  file_length = ftell(file);
  rewind(file);
  buffer = (char*)malloc(file_length);
  res = fread(buffer, 1, file_length,file);
  if (res == 0){
    fclose(file);
    perror("This file could not be read. Check if file exists or correct
        path is given.");
  }
  else{
    fclose(file);
    return buffer;
  }
}


void parse(char* text, char* delimeter){
  char* res;
  res = strtok(text, delimeter);
  while (res != NULL){
    printf("%s\n", res);
    res = strtok(NULL, delimeter);
  }
}


char* sim(char* filename1, char* filename2){
  //Open all files for reading
  FILE* file1 = fopen(filename1, "r");
  FILE* file2 = fopen(filename2, "r");
  //Initialize the output buffer
  char* outputBuff;
```

```
//Check for any NULL pointer received when attempting to open
if (file1 == NULL || file2 == NULL){
  exit(1);
}
//Get the length of each file
fseek(file1, 0, SEEK_END);
int file1Len = ftell(file1);

fseek(file2, 0, SEEK_END);
int file2Len = ftell(file2);

//Allocate space for the output buffer depending on which file is
    bigger
if (file1Len > file2Len){
   outputBuff = (char*) malloc(file1Len + 1);
}
else{
   outputBuff = (char*) malloc(file2Len + 1);
}
printf("\n");
//Reset the file pointer for each to the beginning of the file
rewind(file1);
rewind(file2);

char* lineBuff1 = NULL;
char* lineBuff2 = NULL;
size_t len1 = 0;
size_t len2 = 0;
int res1;
int res2;


while((res1 = getline(&lineBuff1, &len1, file1)) != -1){

  while((res2 = getline(&lineBuff2, &len2, file2)) != -1){

    if (strcmp(lineBuff1, lineBuff2) == 0){

      strcat(outputBuff, lineBuff2);
      free(lineBuff2);
    }

  }
  rewind(file2);
  free(lineBuff1);
}

fclose(file1);
fclose(file2);
```

```
  return outputBuff;

}

void find(char* file, char* string){
  char* temp;
  int lineNum = 1;
  int foundString = 0;

  char* line = strtok(file, "\n");
  while(line != NULL){
    //printf("%s\n", line);
    temp = line;
    if((strstr(temp, string)) != NULL){
      printf("Match on line: %d\n", lineNum);
      printf("\n%s\n", temp);
      foundString++;
    }
    lineNum++;
    line = strtok(NULL, "\n");
  }
  if(foundString == 0){
      printf("\nNo match found\n");
  }
}

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

  char *buffer;
  buffer = read_csv("test_read_csv.txt");

}
#endif
```

9.8 *cst_class_roster.csv*

```
Name, UNI, School, Year, Grade
Tahsina, ts2931, BC, 2019, B
Tasfia, ta2020, CC, 2020, A
Rida, ra2005, CC, 2020, A
Sam, sd2345, SEAS, 2019, A
Callie, cg1010, GS, 2021, C
```

```
Alia, ab1012, BC, 2022, B
```

9.9 *plt_class_roster.csv*

```
Name, UNI, School, Year, Grade
Tahsina, ts2931, BC, 2019, B
Rida, ra2005, CC, 2020, A
Sam, sd2345, SEAS, 2019, A
Callie, cg1010, GS, 2021, C
```

9.10 *test_read_csv.csv*

```
Name, Year, School, GPA
Ana, 2022, SEAS, 4.0
Mona, 2020, BC, 3.7
Sarah, 2018, CC, 3.3
Holden, 2021, SEAS, 2.7
```

9.11 *test_read_csv2.csv*

```
Name, Year, School, GPA
Mona, 2020, BC, 3.7
Sarah, 2018, CC, 3.3
Holden, 2021, SEAS, 2.7
Ana, 2022, SEAS, 4.0
```

9.12 *test_sim.csv*

```
Name, Year, School, GPA
Mona, 2020, BC, 3.7
Tabara, 2020, SEAS, 4.0
Tahsina, 2019, BC, 3.0
Sarah, 2018, CC, 3.3
Holden, 2021, SEAS, 2.7
```

9.13 *test_sim2.csv*

```
Name, School, Finals, Year
Rida, BC, 5, 2019
```

```
Nush, CC, 7, 2018
Tahiya, SEAS, 10, 2020
```

9.14 testall.sh

```
#!/bin/sh

# Regression testing script for ProCSV
# 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 procsv compiler. Usually "./procsv.native"
# Try "_build/procsv.native" if ocamlbuild was unable to create a
    symbolic link.
PROCSV="./procsv.native"
#PROCSV="_build/procsv.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] [.pc 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/.pc//'‘
    reffile=‘echo $1 | sed 's/.pc$//'‘
    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 ${basename}.out" &&
    Run "$PROCSV" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">"
        "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "printbig.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/.pc//'`
    reffile=`echo $1 | sed 's/.pc$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$PROCSV" "<" $1 "2>" "${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 [ ! -f printbig.o ]
then
    echo "Could not find printbig.o"
    echo "Try \"make printbig.o\""
    exit 1
fi

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

9.15 Makefile

```
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : all
all : procsv.native printbig.o

.PHONY : procsv.native
procsv.native :
   ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
      procsv.native

# "make clean" removes all generated files

.PHONY : clean
clean :
   ocamlbuild -clean
   rm -rf testall.log *.diff procsv scanner.ml parser.ml parser.mli
   rm -rf printbig
   rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx procsv.cmx

procsv : $(OBJS)
   ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis
       $(OBJS) -o microc

scanner.ml : scanner.mll
   ocamllex scanner.mll

parser.ml parser.mli : parser.mly
```

```
   ocamlyacc parser.mly

%.cmo : %.ml
   ocamlc -c $<

%.cmi : %.mli
   ocamlc -c $<

%.cmx : %.ml
   ocamlfind ocamlopt -c -package llvm $<

# Testing the "printbig" example

printbig : printbig.c
   cc -o printbig -DBUILD_TEST printbig.c

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and
    parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
procsv.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
procsv.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo

# Building the tarball

TESTS = add1 arith1 arith2 arith3 fib for1 for2 func1 func2 func3 \
    func4 func5 func6 func7 func8 gcd2 gcd global1 global2 global3 \
    hello if1 if2 if3 if4 if5 local1 local2 ops1 ops2 var1 var2  \
    while1 while2 printbig mod inc dec float print_float print_string \
    read_csv sim perse

FAILS = assign1 assign2 assign3 dead1 dead2 expr1 expr2 for1 for2 \
    for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 func8 \
    func9 global1 global2 if1 if2 if3 nomain return1 return2 while1 \
    while2 float print_float print_string read_csv sim perse

TESTFILES = $(TESTS:%=test-%.pc) $(TESTS:%=test-%.out) \
      $(FAILS:%=fail-%.pc) $(FAILS:%=fail-%.err)

TARFILES = ast.ml codegen.ml Makefile _tags procsv.ml parser.mly README \
        scanner.mll semant.ml testall.sh printbig.c arcade-font.pbm
            font2c \
   $(TESTFILES:%=tests/%)

procsv-llvm.tar.gz : $(TARFILES)
   cd .. && tar czf procsv-llvm/procsv-llvm.tar.gz \
      $(TARFILES:%=procsv-llvm/%)
```