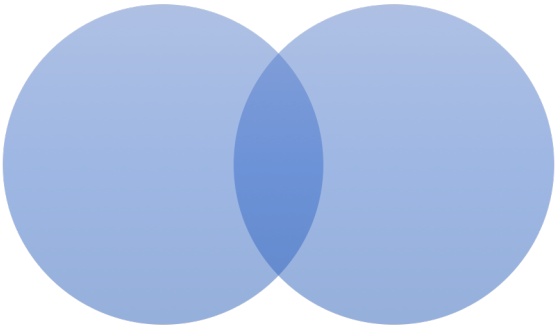
SetC: Final Report

Heather Preslier (hnp2108)



Contents

1 Introduction

The SetC language is a statically inferred language inspired by brevity and ease of set theoretic notation. Its goal is to simplify the formulation of complex algorithms by (1) using a concise language that mirrors set notation and (2) by removing the need for type declarations so as to abstract away some of the implementation details. It also aims to simplify the handling and manipulation of sets.

SetC has functionality for general purpose programming as well as set construction, manipulation and operation. The SetC language has unique set theoretic notations present in constructs such as the set theoretic iterator (similar to a for loop) and cardinality operator, while the statically inferred typing removes the need for all type declarations.The programs developed in this language are compiled into LLVM IR.

2 Language Tutorial

2.1 Setup

The language has been developed in OCaml which needs to be installed to be able to use the compiler. The best way to install is through OPAM(OCaml Package Manager). Using OPAM, OCaml and related packages and libraries can be installed. Follow the below commands for the basic setup.
Note: The version of the OCaml llvm library should match the version of the LLVM system installed on your system.

```
$ sudo apt-get install -y ocaml m4 llvm opam
$ opam init
$ opam install llvm.3.6 ocamlfind
$ eval `opam config env`
```

2.2 Getting Started

Inside the *setc* directory, type `make`. This creates the SetC to LLVM compiler, **setc.native**, which takes as input a SetC file with *.sc* extension and outputs LLVM code. The following is included to demonstrate this process:

Input: `test-hello.sc`

```
def main() {
  print("hello world");
}
```

The **setc.native** executable can then be used to compile and run this code using the following:

```
$ ./setc.native < test-hello.sc | lli
>> hello world
```

2.3 Simple Example

SetC is a statically inferred language that is syntactically similar to C without the type declarations. The one requirement is every executable program must include a **`main`** function which serves as the entry point into the program. The following is a simple code sample that demonstrates some of the setc features:

```
def main() {
  a = 4;
  c = 5;
  d = add(a, c);
  print(d);
}
/* adds two numbers together */
def add(a, b) {
  return a + b;
}
```

This yields the following output:

```
>> 9
```

The above program demonstrates the mandatory **main** function, type inference of variables, functions, and parameters, and the use of the built-in print function.

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Spaces, tabs, and newlines can be used interchangeably to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

3.1.2 Comments

Similar to the C language, characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest, and they do not occur within string literals. Comments are ignored by the syntax; they are not tokens.

3.1.3 Identifiers

All identifiers must begin with a lowercase alphabetic letter, followed by alphanumeric characters or the underscore _ character. Identifiers can be any length and letter case is significant.

3.1.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

`def`	`while`
`return`	`break`
`if`	`true`
`else`	`false`

3.1.5 Literals

Literals are constant values of some built-in type. There are four type of literals: ints, floats, bools, and strings.

literals:

integer_literal
float_literal
string_literal
true
false

3.1.5.1 Integer Literals

Integer constants are a sequence of digits that are taken to be decimal.

3.1.5.2 Floating Literals

A floating constant consists of an integer part, a decimal point, and a fractional part. The integer part may be absent if the decimal point and fractional part are present. The fractional part may be absent if the integer part and decimal point are present.

3.1.5.3 String Literals

A string literal consists of a collection of characters enclosed in double quotes as in "...". It is not possible to index through a statically declared string nor is it possible to manipulate the data contained in the string.

3.1.5.4 Boolean Literals

Boolean literals are represented by the keywords `true` and `false`.

3.2 Meaning of Identifiers

Identifiers can be function names, variable names, or other such declared expressions. An object, sometimes called a variable, is a location in storage, and its interpretation depends on what its type is. The type determines the meaning of the values found in the identified object. A name also has a scope, which is the region of the program in which it is known, and a linkage, which determines whether the same name in another scope refers to the same object or function.

3.2.1 Basic Types

The fundamental types present in SetC are integers, floats, bool, strings, and void, however void is only ever used implicitly when declaring an empty set (discussed below in 3.2). All basic types are immutable data types.

3.2.2 Complex Types

A set is a mutable collection constructed from the fundamental types. All elements of a set must be of the same type. A void type set is declared with empty brackets and nothing in between as follows:

```
set = [];
```

The (nonexistent) value of a void object may not be used in any way. Accessing elements or applying any set operations on a void set will result in undefined behavior.

3.3 Type Inference

The SetC language is a statically inferred language. Identifiers do not have to be preceded by a type. Once the type is inferred, it is bound to that identifier for the rest of the identifier's lifetime.

3.3.1 Variable Inferences

For variable identifiers, all that is required for initialization is an assignment to an expression. The type of the expression it is assigned to becomes the type of the identifier.

There is one subtlety with sets. Sets can be declared by either listing the set contents explicitly within square brackets or by declaring an empty set as shown below.

```
set1 = [1, 2, 3];
set2 = [];
```

The empty set has no type associated with it until its **first** use. For instance, if it is used with the append "+" operator for sets or the in "?"

operator for sets, it's type will be inferred from that usage. The following code snippets will infer that variablesa is of type `int`.

```
a = [];
3 ?  a;
```

3.3.2 Function Inference

For functions, the keyword `def` followed by the identifier is needed. A function's type is determined by its return type. If there are multiple return types and the types do not match, a return type mismatch error is raised.

3.4 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The grammar given in Section 3.8 incorporates the precedence and associativity of the operators.

expression:
    logical-OR-expression
    expression , logical-OR-expression

3.4.1 Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers and constant literals. Forms enclosed in parentheses or brackets are also categorized syntactically as atoms.

atoms:
    literals
    identifier
    (expression)
    set

3.4.1.1 Identifiers

An identifier's type is inferred by its assignment. A parenthesized expression is a primary expression whose type and value are identical to the type and value of the inner expression.

3.4.1.2 Literals

A literal is an atom. Its type depends on its form discussed in Section 3.1.5.

3.4.1.3 Parenthesized forms

A parenthesized expression is an atom whose type and value are identical to those of the unadorned expression.

3.4.1.4 Sets

A set is a possible empty series of expressions enclosed in square brackets.
set:
    [*expression_list_opt*]

expression_list_opt:
    / nothing */*
    expression_list

expression_list:
    expression
    expression_list, expression

A possibly empty list of expressions enclosed in square brackets yields a new set object. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the set object in that order.

3.4.2 Primaries

Primary functions are the most tightly bound operations in the language. Primary functions group left to right.

primary:
    atom
    index
    call
    slicing

3.4.2.1 Indexing

An index selects an item of a set:

index:
    identifier[expression]

An identifier followed by an expression in square brackets is a subscripted set reference. The identifier must have set type and the expression enclosed in brackets must have integral type.

3.4.2.2 Function Calls

A function call is an identifier followed by parentheses containing a possibly empty comma-separated list of arguments.

call:
    identifier (actuals_opt)

actuals_opt:
    / nothing */*
    actuals_list

actuals_list:
    expression
    actuals_list, expression

All argument expressions are evaluated before the call is attempted. If there are more positional arguments than there are formal parameter slots, an exception is raised.

3.4.2.3 Slicing

A slicing selects a range of items in a set. The syntax for a slicing is as follows:

slicing:
    identifier[expression : expression]

Slicings return a new list with only the elements within the range specified by the expressions. The identifier must be of set type and both the expressions must evaluate to integer type.

3.4.3 Unary Operators

There are three unary operators: the `#`, `!`, and – characters.

unary-expression:
    primaries
    unary-operator primaries

unary-operator: one of
    ! # -

The `#` is the cardinality operator used with sets; it returns the length of the set. The `!` operator is used for logical negation of boolean expressions only. The – operator is used for arithmetic expressions or literals and the result is the negative of its operand.

3.4.4 Set Operators

The binary in operator ? takes two operands.

set-expression:
    unary-expression
    set-expression ? unary-expression

The second expression must be of type set and the first expression must be of whatever the type of the set elements of the second expression is. This operator returns a boolean value representing the presence or absence of the first expression in the second.

3.4.5 Multiplicative Operators

The multiplicative operators `*`, `/` and `%` group left to right.

multiplicative-expression:
set-expression
multiplicative-expression `*` *set-expression*
multiplicative-expression `/` *set-expression*
multiplicative-expression `%` *set-expression*
multiplicative-expression `&` *set-expression*

The operands of `/`, and `%` must be of arithmetic type. The binary operator `/` denotes division. The binary operator `%` denotes modulo.

The operands of `&` must be of set type. The binary operator `&` then denotes the union of the two sets. That is, it returns a new set of all the elements in the first set and the second set (with no repetitions). The operands of `*` can be of either arithmetic or set type. If the operands are of arithmetic type, the binary operator `*` denotes multiplication. If the operands are of set type, the type of the elements within both sets must be of the same type. The `*` operator then denotes the intersection of these two sets.

3.4.6 Additive Operators

The additive operators + and − groups from left to right.

additive-expressions:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression − *multiplicative-expression*

The operands of + and − can be of either arithmetic or set type.

If the operands are of arithmetic type then the result of the operation is the standard arithmetic result. The + operator denotes addition and the operation results in the sum of the operands. The − operator denotes subtraction and the operation results in the difference of the operands.

If the operands are of set type the + operator becomes the append/concatenation operator. That is, it returns a new list with all the elements of the first set followed by all of the elements of the second set. The − operator becomes the set difference operator; it returns a new list of all the elements in the first set that are not in the second set. Both sets' elements must be of the same type.

3.4.7 Relational Operators

The relational operators < (less), > (more), <= (less or equal) and >= (more or equal) group from left to right.

relational-expression:
    additive-expression
    relational-expression < *additive-expression*
    relational-expression > *additive-expression*
    relational-expression <= *additive-expression*
    relational-expression <= *additive-expression*

The operands of relational operators must be of the arithmetic type. An relational operation evaluates to a boolean expression: true or false.

3.4.8 Equality Operators

The equality operators == (equal to) and != (not equal to) groups from left to right.

equality-expression:
    relational-expression
    equality-expression == *relational-expression*
    equality-expression != *relational-expression*

The operands of the equality operators can be of any type other then set type: boolean, string, int, and float. The equality operators have a lower precedence than the relational operators. An equality expression evaluates to a boolean expression: true or false.

3.4.9 Logical AND Operator

The logical AND operator `&&` groups from left to right.

logical-AND-expression:
    equality-expression
    logical-AND-expression `&&` *equality-expression*

The operands of the `&&` operator must evaluate to boolean type. The expression returns true if both operands evaluate to true and returns false otherwise. If the left hand operand evaluates to false, the right hand operand is not evaluated. It returns a value of type boolean.

3.4.10 Logical OR Operator

The logical OR operator `||` evaluates from left to right.

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression `||` *logical-AND-expression*

The operands of the `||` operator must evaluate to boolean type. The expression returns true if either of the operands evaluate to true and returns false otherwise. If the left hand operand evaluates to true, the right hand operand is not evaluated. It returns a value of type boolean.

3.5 Statements

A statement describes an action to be performed (usually sequentially). Statements are executed for their effect, and do not have values. They can be simple or compound; simple statements only enclose themselves. They fall into several groups.

statement:
    assign_statement
    expression_statement
    compound_statement
    select_statement

iteration_statement
jump_statement

3.5.1 Assignment Statements

Assignment statements are used to bind or rebind names to values and to modify attributes or items. An assignment statement therefore requires that the identifier is one that is mutable or modifiable.

assign_statement:
identifier = expression;
identifier[expression] = expression;

If the first assignment statement is used on a **new** identifier, it acts as an initialization for the identifier. When the = operator is used, the identifier will be replaced by the right hand expression and the type of the identifier will be the type of the right hand expression. For the rest of the variable's lifetime, it will be bound to this type. If the first assignment statement is used to rebind values (the identifier already exists), it requires that both the identifier and expression evaluate to the same type.

If the second assignment statement is used, the identifier must be of set type, the expression enclosed in brackets must be of integer type, and the right hand side expression must be whatever the type is of the elements in the set the identifier is referring to.

3.5.2 Expression Statement

Expression statements can be used for computing or writing values or for function calls. Expression statements are completed using the semicolon `;` operator.

expression_statement:
expression;

3.5.3 Compound Statement

A compound statement consists of simple statements, or other compound statements.

compound_statement:
{ *statement_list* }

statement_list:
/ nothing */*
statement_list statement

Compound statement declarations retain scope within their encompassing curly braces. If identifiers of the same name are declared earlier in scope, only the most recent declaration identifier will be visible.

3.5.4 Selection Statement

Selection statements are made up of `if, else` statements. They are used to evaluate an expression, and depending on this evaluation, execute a specific action. These selection statements are applied similarly in SetC as they are in C.

select_statement:
`if` *(expression) statement*
`if` *(expression) statement* `else` *statement*

3.5.5 Iteration Statements

Iteration statements have two varieties: statements declared with `while` and statements using set theoretic notation.

iteration_statement:
`while` *(expression) statement*
(constraints_list expression_opt) statement

expression_opt:
/ nothing */*
| *expression*

constraints_list:
constraints

constraints_list, constraints

constraints:
expression < *identifier* < *expression*
expression <= *identifier* <= *expression*
expression < *identifier* <= *expression*
expression <= *identifier* < *expression*
expression > *identifier* > *expression*
expression > *identifier* >= *expression*
expression >= *identifier* >= *expression*
expression >= *identifier* > *expression*

While loops are contingent on the structure of the conditional statement inside the parentheses that follow.

For set theoretic iteration, constraints must be inside the parentheses. The identifier used in these constraints are the loop variables. Encompassing this variable with < or <= operators will increment the variable and > or >= operators will decrement the variable. Using < or <= with > or >= will raise an iteration error.

3.5.6 Jump Statements

Jump statements transfer control unconditionally.

jump-statement:
`break` ;
`return` *expression_opt* ;

A `break` statement may appear only in an iteration statement and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the `return` statement. When return is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears. Flowing off the end of a function is equivalent to a return with no expression. In either case, the returned value

is undefined.

3.6 Function Definitions

Functions must be declared and defined as follows:

function_declarations:
    `def` *identifier (params_opt)* { *statement_list* }

params_opt:
    param_list

param_list:
    identifier
    param_list, identifier

A function may return any type: arithmetic, set, or boolean. The parameters of the function are a possibly empty comma separated list of identifiers only. The type of the function is inferred by the function's return type as discussed in section 3.3.2.

3.7 Scope

Identifiers can fall into two different namespaces that do not interfere with one another: functions and variables. Therefore, the same identifier may be used as a function name and a variable name.

Functions may not have the same identifier.

Variables in different scopes can have the same identifier. However, variable identifiers in nested scopes can only refer to one object. When a variable identifier is used within a scope where the variable has already been defined in either an enclosing scope or the same scope, it refers to the outermost identifier. The scope of the identifiers are defined within the region of code in which they are declared, denoted using curly braces { and }. SetC consists of local and globally declared identifiers. The lifetime of the identifier is determined by its scope.

3.8 Grammar

program:
    declarations

declarations:
    declarations variable_declarations
    declarations function_declarations

variable_declarations:
    identifier = expression;

function_declarations:
    `def` *identifier (params_opt)* { *statement_list* }

params_opt:
    param_list

param_list:
    identifier
    param_list, identifier

statement_list:
    / nothing */*
    statement_list statement

statement:
    expression_statement
    select_statement
    assign_statement
    compound_statement
    iteration_statement
    jump_statement

expression_statement:
    expression;

select_statement:

`if` (expression) statement
`if` (expression) statement `else` statement

assign_statement:
identifier = expression;
identifier[expression] = expression;

compound_statement:
{ *statement_list* }

iteration_statement:
`while` (expression) statement
(constraints_list expression_opt) statement

jump-statement:
`break` ;
`return` expression_opt ;

expression_opt:
/ nothing */*
| *expression*

constraints_list:
constraints
constraints_list, constraints

constraints:
expression < identifier < expression
expression <= identifier <= expression
expression < identifier <= expression
expression <= identifier < expression
expression > identifier > expression
expression > identifier >= expression
expression >= identifier >= expression
expression >= identifier > expression

expression:
logical-OR-expression

expression , logical-OR-expression

logical-OR-expression:
logical-AND-expression
logical-OR-expression || logical-AND-expression

logical-AND-expression:
equality-expression
logical-AND-expression && equality-expression

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

relational-expression:
additive-expression
relational-expression < additive-expression
relational-expression > additive-expression
relational-expression <= additive-expression
relational-expression <= additive-expression

additive-expressions:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression − multiplicative-expression

multiplicative-expression:
set-expression
multiplicative-expression ⋆ set-expression
multiplicative-expression / set-expression
multiplicative-expression % set-expression
multiplicative-expression & set-expression

set-expression:
unary-expression
set-expression ? unary-expression

unary-expression:
    primaries
    unary-operator primaries

unary-operator: one of
    ! # -

primaries:
    atoms
    index
    slicing
    call

index:
    identifier[expression]

slicing:
    identifier[expression : expression]

call:
    identifier (actuals_opt)

actuals_opt:
    / nothing */*
    actuals_list

actuals_list:
    expression
    actuals_list, expression

atoms:
    literals
    identifier
    (expression)
    set

set:
    [*expression_list_opt*]

expression_list_opt:
/ nothing */*
expression_list

expression_list:
expression
expression_list, expression

literals:
integer_literal
float_literal
string_literal
true
false

4 Project Plan

4.1 Process

4.1.1 Planning

Before beginning the project, major milestones for building the SetC compiler were made. Every week throughout its duration, short term goals were set to reach the next milestone. Below is a table of the milestones set for the project:

Milestones
Complete Project Proposal
Complete Language Reference Manual
Complete parser and scanner files
Get Hello World printing
Implement Expressions
Implement Statements
Implement Sets
Implement Standard Libraries

The first milestone was to create the parser and scanner files followed by getting hello world to print. I chose this order because I felt it was better to have minimal functionality across all files, which I would achieve by getting "hello world" to print, then to implement a lot in one particular file without others yet working. Implementing expressions, statements, and sets were next, followed by writing the standard libraries.

4.1.2 Testing

As detailed in the Test Plan section, unit tests were written after each new feature was added to verify it was working correctly. The tests usually incorporated earlier functionality as well. At various stages in the project, black box integration tests were added to the test suite to make sure that everything was working correctly together as well.

To pinpoint the errors that were found in the generated code, different debugging techniques were used. Depending on the type of bug, debugging could have been done either by sifting through the LLVM IR generated code and comparing it to the generated LLVM IR of a similar program written in C or by consulting the visual representations of the ast and sast intermediary data structures. Both proved extremely helpful in finding and fixing errors.

4.2 Timeline

A detailed timeline of the project is given below. Testing was performed throughout the duration of the project. The code writing portion of the project started a little after planned but hard work during the month of April quickly brought me back up to speed. There were rarely ever definitively "completed" milestones, as most files of the compiler were restructured and changed until the end.

Date	Milestone
February 8	Project Proposal complete
February 22	Language Reference Manual complete
March 8	First commit
March 22	Minimal Functionality in scanner and parser
March 23	Hello World runs
March 25	Scanner and parser completed
March 28	Testing script created
April 03	Created the sast file
April 04	Working on the semant file
April 05	Working on the codegen file
April 06	Expressions completed; exception file created
April 11	Statements completed
April 13	Restructuring of codegen; parameters working
April 15	Sets compiling
April 22	Linking files with standard library
April 24	New implementation of sets
April 25	Adding SetC library functions; sets completed
May 5	Adding C library functions
May 10	LRM and Final Report complete

4.3 Roles and Responsibilities

The responsibilities I had for this project included the compiler front end (lexer and parser), semantics, code generation, documentation, SetC and C libraries, AST and SAST visualization, and unit and integration testing.

4.4 Software Development Environment

The programming and development environment for the project is as follows:

- Environment: I worked in a virtual machine running ubuntu. I used vim to code and git version control to organize and track changes.
- Language: The programming language I used for building the compiler was OCaml version 4.01.0. I used two program generators, Ocamlyacc and Ocamllex, to compile the lexical analyzer (scanner.mll) and parser (parser.mly).

4.5 Project Log

This project log shows a history of 106 commits starting from March 8th and ending May 9th.

```
commit 8b17572856369435df4abc47fe1b5cb60e9ecb2f
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue May 9 20:53:20 2017 -0400
commit c1072586f8e55edaf08786289a4da7c224a5836e
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue May 9 12:47:17 2017 -0400
commit 8a8eaf3167827cd2c5c7218b7f65475b3a2ac14b
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue May 9 12:41:46 2017 -0400
commit 2c0ad93fc21acc8a7442b7c660444b26f8c084e1
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue May 9 12:36:05 2017 -0400
commit 9bd3ef198c00ee472d6291bf98b66fb5d5d71795
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue May 9 11:45:28 2017 -0400
commit d676383951fed511d0b29861d33f77fc5c3aff5b
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue May 9 11:03:14 2017 -0400
commit 44c10960b1a06b11b71bb4ac57783dd99aab7245
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri May 5 23:08:50 2017 -0400
commit 40d3d6ece5930d5a17bf605c92180047e9eb47ab
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri May 5 23:08:09 2017 -0400
commit 8b8effe099c3456cfd42d67c86580ad8491132f2
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri May 5 22:56:59 2017 -0400
commit 6ec8ff4a0633978c657bb82578cc21a25cde59a3
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu May 4 22:17:02 2017 -0400
commit 2249dc3cc90d5ce986853003a4d9e0bd88fceb4b
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sun Apr 30 00:03:10 2017 -0400
commit b90e1628ff7f265bba1725700a2655ef2dbbc6dd
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 29 23:18:42 2017 -0400
```

```
commit 1cc95a78b92b799347d7fc240139f37f1125a1a7
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 29 23:17:20 2017 -0400
commit 25eb256e09ead14bff38bd7e4f2315146c8d5d72
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 29 22:47:09 2017 -0400
commit 12e1671eb459728bdeacf8e0a5f2c4c2dd15544a
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 23:35:32 2017 -0400
commit fb6d0b847102836ee15bd76852fbf5b67aab1c75
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 22:58:28 2017 -0400
commit 7da2e9ffe9cb4b99f6174c5793e3f65f79308bf6
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 22:37:31 2017 -0400
commit 178be2a22dde01b9f466408bed3ba5f00172ba8e
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 22:13:47 2017 -0400
commit e96562d64aac43e64b5a22890ad79321d68a04fa
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 22:10:37 2017 -0400
commit 815c5d5ef7297416523c643d1e2d0ac35970076f
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 21:59:45 2017 -0400
commit 528906501d1cec6b056f1dd9766037402305a780
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 21:40:50 2017 -0400
commit 70ace4a1b7af51c7e46f010c0d0b51b20dbd447f
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 19:51:36 2017 -0400
commit 4d774eb2b7243b0b13a0bb9bb58f46ed42b51537
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 19:27:50 2017 -0400
commit 1498076262a64038adb5c3ffbb5194998c92f8c3
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 18:50:11 2017 -0400
commit 39f56fdda60222206b8304a0be38eb21c04e1baf
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 15:01:53 2017 -0400
commit ac4140d90a9b7ada19fe4cd47f87ae1bafb03328
```

```
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 14:55:41 2017 -0400
commit c3328e4ba95210561b78d77acae4b4a62aa7d565
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 14:34:18 2017 -0400
commit 63e48152e92c88e50b699fc8234f07ff2828df06
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 14:24:02 2017 -0400
commit 1db8a450e8678eb6aa59b3fc25065520f5184d25
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 25 01:52:03 2017 -0400
commit c12c20c89bcd0826fad4049afe45588ae484fbdd
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 24 23:55:35 2017 -0400
commit ffb4b133748b74a5ae512f38f860a245d8cfa5da
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 24 23:45:45 2017 -0400
commit b2f15be599294aceaee2d568c7698f25bc28a25f
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 24 23:39:44 2017 -0400
commit c00a90f82c0a51774ba02dc203930e4c48e2b7cb
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 24 22:04:41 2017 -0400
commit 01a0c290e1552c20bb38a57ec42f7dba0bfd22eb
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 24 17:41:32 2017 -0400
commit 9671abc591914f3e3d56761a13f64ad8ed6f1fc7
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 24 15:32:45 2017 -0400
commit 336e4056be87b383ea9748021c78536843aec942
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 24 14:56:45 2017 -0400
commit 706c3cb0555ec9c217061d38de04e7a71188cb0c
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 22 15:13:04 2017 -0400
commit 5fd9c95c52b21bc79fb0d21df1b9a438e278a749
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 22 14:30:00 2017 -0400
commit 8e3e3a328667aa9def84f8d2c68a31cf33a7fc60
Author: Heather Preslier <heatherpreslier@hotmail.com>
```

```
Date:   Sat Apr 22 01:05:10 2017 -0400
commit e9d4d5dafe4a6bbc7071a85b113cec794f070eb7
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Apr 21 21:20:43 2017 -0400
commit e9b2f3e81b9ff697c85195c5b15ead8ddad525b7
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 20 00:58:31 2017 -0400
commit 70704b0bdc7a61943d3eac59670b30c60c07168e
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 20 00:15:07 2017 -0400
commit cd0c8609a7bd08a90a05279f35f07667aa536e4b
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 19 22:11:37 2017 -0400
commit 0157ac28fb9119c5ce8a448280153790a5e91cb9
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 19 00:25:41 2017 -0400
commit 443d90f2fc66d09c892994e7b46bb828301a3581
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 18 23:25:34 2017 -0400
commit 8133232f655cd30044d8067268b9d9ae4d713e7b
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 18 22:29:38 2017 -0400
commit ecd12a9242f2cf68c5b1f359eadc46a16993b9b5
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 18 19:45:43 2017 -0400
commit 8bdfa7d8ba12b6eec248e24f188ee00b80c361e1
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 18 01:23:07 2017 -0400
commit f24b8cbfd4fe0d614004c53a9021e5b7919ce409
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 15 02:20:33 2017 -0400
commit 9f3b231c1c6b6dee31748fc8e5fdda37e549aaad
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 15 01:08:26 2017 -0400
commit 36f6bca6371f1f94c6a32cfc8ddd1a0aa0cd2027
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 15 00:55:21 2017 -0400
commit a7e82c76f23ff54405a36f8efe2fd19301d21666
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 15 00:01:10 2017 -0400
```

```
commit f2dfef568cd4f52140447eb2333f3b56ab6aabd2
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Apr 14 19:18:19 2017 -0400
commit 35720a296a0bb4674e77e7523b50812a89835ae0
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Apr 14 19:04:37 2017 -0400
commit acf73147e1fef3928da846e268a464a7ea214487
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Apr 14 18:33:13 2017 -0400
commit 0682987b94a0781bf5886ce0d086f7302d42ac04
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Apr 14 13:52:20 2017 -0400
commit ec94364bf3dc3b0822bba967fd8f00b836838da0
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 21:44:36 2017 -0400
commit 47f365ad3697473d26503ef6bee12beee48ca6e2
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 18:18:45 2017 -0400
commit 8311ca77b38c3c96a89c4c049941bdd1343db4fa
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 15:22:48 2017 -0400
commit 2a61576e41dd63055f198ebf720fec55791690da
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 14:59:33 2017 -0400
commit 634c4b5dc1407076c4d602c37b843dd14ea2b6e7
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 12:21:07 2017 -0400
commit e8ce1fe4981c828063a5f0795427b15fdd5c00ed
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 03:59:44 2017 -0400
commit f80199739359 42b30bbea28a7bffe8c884b40479
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 03:28:59 2017 -0400
commit fd3e3487bcba2009a9a89fc4cd9a32ec60c5204a
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 03:22:49 2017 -0400
commit 4d812129069d80f15680043a0d1bdf5ce596ce27
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 02:30:52 2017 -0400
commit 2cc6e2acd114dcc65ad566405e6efad62e7d9a75
```

```
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 01:58:28 2017 -0400
commit e0719da15945ce73a402d3375207e61ddc5696d6
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 13 00:09:18 2017 -0400
commit 353f63b20af5c15894decde19c84a1ceaa55d090
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 12 22:52:04 2017 -0400
commit 0042bae7dc97b702f0362591e3f897d87d075e2e
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 12 14:59:41 2017 -0400
commit 6ab8b3bf75b856040c79bb457b2a89e12225ba17
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 12 02:07:34 2017 -0400
commit b796fe62724768db346aa39dbfb09602d079ab97
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 11 19:04:12 2017 -0400
commit 26ca35838d82cb98c47115dff440afcd64dd9b23
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 10 19:29:41 2017 -0400
commit 9f96541c1072bd176a21d0b45ba06f9e56ef079e
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 10 18:45:20 2017 -0400
commit 8d61217b475689ef45e521eed28e5df0d939c10a
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 10 18:22:04 2017 -0400
commit cb6d542a8ce598f6ea26647971fe03650f6de08f
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Apr 8 18:24:24 2017 -0400
commit 0fa0716008fc00359072668a0047b10fab2037c5
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 6 17:58:27 2017 -0400
commit 4a07817810ab97a433c38b5ba24ee52143189e2d
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 6 15:51:36 2017 -0400
commit 6549f277511f251c6d221304b1a36f7d35c1f89e
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Apr 6 00:06:35 2017 -0400
commit 2bfe409947ec2e92bf4a51066c19a21b1618e453
Author: Heather Preslier <heatherpreslier@hotmail.com>
```

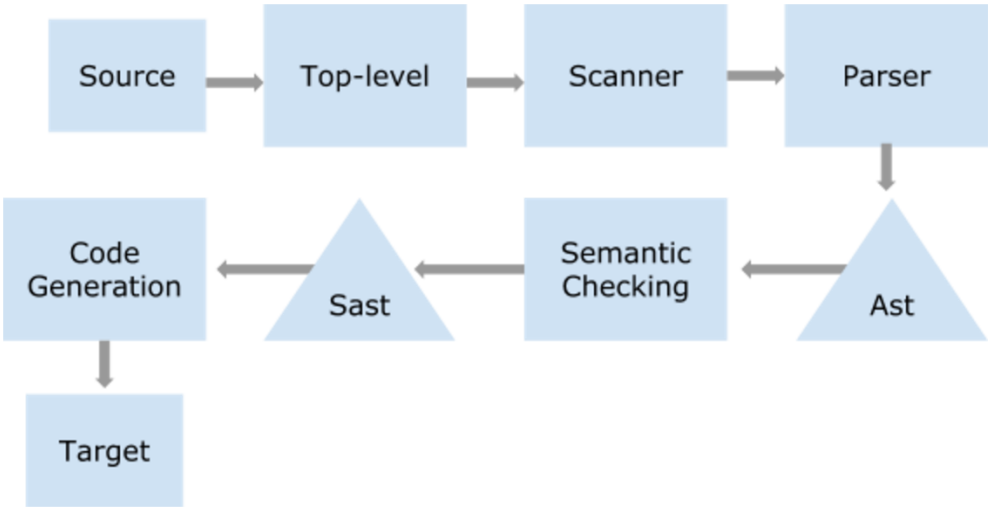
```
Date:   Wed Apr 5 21:39:22 2017 -0400
commit 6380a9bb9f05822525ef002b69421b56532f22ce
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 5 21:07:39 2017 -0400
commit 31f3ed59cd42b30de9de1dc30d831b815b8c5965
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 5 16:50:30 2017 -0400
commit df00cc62d21eb436161cfca452624c3129a98af2
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Apr 5 14:23:18 2017 -0400
commit 14196e63e964dcd7f356411f026776d13d41c620
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 4 23:20:26 2017 -0400
commit 656229986586c67e3b498eddf944d6d0f63c524c
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Apr 4 21:45:48 2017 -0400
commit 89d61579eb8a43b9065869696bb9693a5b6e5b4e
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Mon Apr 3 20:39:05 2017 -0400
commit d4c2a2018bdc71e7d54fa62458de39e2b4709660
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Mar 28 20:12:24 2017 -0400
commit 3a46626fe7858bfc8a7948499291d35b42d9c122
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Mar 28 19:40:57 2017 -0400
commit 493b038d5ad66fcc2aa0f9c4768486d2b767d3d0
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Mar 28 18:55:11 2017 -0400
commit 6f57ba306e1bf26c482c3e194b8da6fa66270022
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Tue Mar 28 18:31:06 2017 -0400
commit 6a8d6604ff321190a38be9a9d7b1cea233fa005b
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Sat Mar 25 00:05:34 2017 -0400
commit ef7f871762da2af4e37b905f500d3449992d97c8
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Mar 24 20:41:46 2017 -0400
commit b7c216ba9d89a3c6e9ed3ccef7a06835bd23cda2
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Mar 24 20:07:25 2017 -0400
```

```
commit 30b91e98d48f58fbcaab7d65bcb4be9303ea3208
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Mar 24 17:43:19 2017 -0400
commit cc26d0b0798a161557f7563c1df9e12f9464b9af
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Mar 23 21:04:58 2017 -0400
commit a5cd45612019e5305b5088a2f003ef0d0c625971
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Mar 23 19:55:42 2017 -0400
commit 31b2f5e2f75dcd91fb586cb13ccaa3ff37857679
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Mar 23 19:20:21 2017 -0400
commit c2e98a2b76a9f3afc2d9e846f26a05b222912add
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Mar 23 19:13:36 2017 -0400
commit 8a108666218e343b2b04e2987ceebef9cafe77b0
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Mar 23 18:21:26 2017 -0400
commit 5c7641c1441c9d1014859badcce4e9688628b56b
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Mar 23 16:09:38 2017 -0400
commit 0ce6c87499cd2f5f124513ee89ec067c6c9ce74a
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Thu Mar 23 15:58:34 2017 -0400
commit a0c40831c7ac25d62fe967734598bc5048c95c37
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Mar 22 19:02:56 2017 -0400
commit 89a1cb1f089ab1b3aadc2cdc0bed4742a36dbc59
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Mar 22 17:10:54 2017 -0400
commit e3fba53d89940738aba3b36f331250151a1770fc
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Mar 10 18:06:44 2017 -0500
commit 80b5022d97f62620f2cf192362773186702cf55c
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Mar 10 17:35:06 2017 -0500
commit edc59297839e9e5b94a5e59f45e573747476132d
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Fri Mar 10 17:19:47 2017 -0500
commit 451faf8c5028835210e185cfebc4608ada632437
```

```
Author: Heather Preslier <heatherpreslier@hotmail.com>
Date:   Wed Mar 8 00:25:00 2017 -0500
```

5 Architectural Design

5.1 Block Diagram



5.2 Top Level File

Filename: `setc.ml`

This is the top-level file, written in OCaml, of the SetC compiler. When a program is run through the `setc.native` compiler, this is the first file the program passes through; it calls functions in the scanner, parser, semant, and codegen files to scan, parse, semantically check, and generate LLVM IR, respectively, and then dumps the module. This file requires that the `stdlib.sc` file be in the current directory as it appends the library to every program that is run through the compiler, so that each has access to the library functions. Depending on the option specified, the output of running the program can be the ast (option -a), the sast (option -s), llvm ir (option -l), or the compiled program (option -c or nothing).

5.3 Scanner

Filename: `scanner.mll`

The scanner, written in OCamlLex, takes the input file (with a ".sc" extension) and tokenizes it into keywords, identifiers and literals. It also disregards (and thus removes) all comments in the input file. If there is anything in the input file that is not syntactically valid, the scanner will throw an error. The tokens created by the scanner are then used by the parser to create the abstract syntax tree.

5.4 Parser and Ast

Filenames: `parser.mly, ast.ml`

The parser, written in OCamlYacc, takes a series of tokens generated by the scanner and, with the grammar defined in the "rules" section of the OCamlYacc file and the types defined in `ast.ml`, generates an Abstract Syntax Tree (AST). If the code is successfully parsed, then it is syntactically correct.

5.5 Semantic Checking and Sast

Filenames: `semant.ml, sast.ml`

The semantic checker, written in OCaml, takes the Abstract Syntax Tree generated by the parser and, with the new types defined in the `sast.ml`, translates the AST into a Semantically Checked Abstract Syntax Tree (SAST).

The semantic checker is responsible for resolving **all** types of the previously typeless functions and variables from the program in the input file. It is also responsible for making sure that all parts of the program (expressions, statements, and functions) are semantically valid.

The semantic checker works by using an environment variable, an aggregator argument to every function within the semant file, that tracks global variables, functions within the file, functions that have been semantically checked, the current local variables for a function, whether the current func-

tion's return type has been set, and the function's return type. This environment variables allowed me to perform full type inferencing.

5.5.1 Type Inferencing

In order to perform type inferencing, I had to think about a few things. First, I realized that it did not make sense to semantically check a function on its own that has not been used yet (unless there are no arguments to the function) and, second, that it did not make sense to semantically check one function after the other. The following code snippet demonstrates these dilemmas:

```
1  def main() {
2    a = 4;
3    b = 6;
4    c = foo(a, b);
5    b =  b + c;
6  }
7  def bar(a, b) {
8    return func(a, b);
9  }
10 def foo(a, b) {
11   return (a + b);
12 }
```

- In regards to the first issue, the function `foo` takes two parameters `a` and `b`. These parameters have no type associated with them initially. Therefore, if we were to semantically check this function first, the expression `a+b` could not be resolved; if it is an integer, set, or float it is valid but if it is a bool or a string it is not. What if we started with main? This would allow us to resolve the types of `foo` as two integers and we could then semantically check it. However, when we moved to check `bar` we would have the same problem as we did initially.
- The second issue can be seen in lines 4-5. If we try to resolve the types in main first and then move to `foo` and `bar`, what will the type of `c` be in line 4? Since we haven't semantically checked `foo` yet, we do not know and, therefore, semantically checking line 5 will not make sense.

If `foo` returns an integer it will work but if foo is any other type it will not. Therefore, we cannot continue to check the rest of main until `foo` has been resolved.

The conclusion of this leads me to my algorithm: I start by semantically checking main and when a function is called pause the semantic checking of the current function, resolve the parameter types to the called function and semantically check the called function. A function's parameter types will then be resolved by the first function that calls it and the first return statement that is checked becomes the return type for the function.

Due to this, it was convenient to optimize out the functions that were not called from the main function (the entry point into the program) or any function that main calls and so forth (i.e. any function that would never be called during the program's entire execution). Therefore, those functions do not get semantically checked (as they are not used during the program).

5.6 Code Generation

Filename: `codegen.ml`

The code generator, written in OCaml, takes the SAST produced by the semantic checker, and generates LLVM IR. This file is mainly responsible for generating LLVM code for all parts of the program (expressions, statements, and function). It is also responsible for building the set type. Sets were originally just pointers but because of the need to constantly have access to the length, I used structs, with a pointer member that hold the address of the first element of the set in memory and an int member that represents the length of the set, to represent them. The codegen file is then also responsible for constantly updating this length field when required so that the length of the set can be updated during run time rather than compile time.

6 Test Plan and Scripts

6.1 Test Suite

There were 103 tests total that were written, some of which were unit tests and others which were integration tests. Immediately, after every compile,

the test script was run to make sure that the latest compile did not break any previous functionality.

6.1.1 Unit Testing

As the compiler was developed, tests were written every time a new feature or component was added to verify that it worked correctly. For each new feature, 1-2 tests were written and added to the *tests* directory.

6.1.2 Integration Testing

Integration tests were added over the course of the development of the compiler. These tests uncovered most of the bugs within the compiler. Most bugs were due to the environment variable not being updated correctly in the semant file. Some, however, seemed to reveal a fundamental problem in the implementation of the code and, thus, required a complete restructuring of the code.

6.2 Test Automation

The `testall.sh` test script in the SetC directory compiles, runs, and links (with the `stdlib.o` library object file and the `strcmp.o` object file) all the files within the *tests* directory. The files must begin with either "test-" or "fail-" and must have the extension ".sc". They must also be accompanied by a file with the same base name and the extension ".out" or ".err" if it starts with a "test-" or "fail-" respectively. The output of the test script is shown below:

```
$ ./testall.sh
>>
test-add...OK
test-add1...OK
test-append-float...OK
test-append...OK
test-append1...OK
test-append2...OK
test-arith1...OK
test-arith2...OK
test-arith3...OK
test-card...OK
```

```
test-card1...OK
test-card2...OK
test-diff...OK
test-el-assign...OK
test-fib...OK
test-for1...OK
test-for2...OK
test-for3...OK
test-found...OK
test-found1...OK
test-func-set...OK
test-func1...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-if...OK
test-if1...OK
test-if2...OK
test-if3...OK
test-if4...OK
test-if5...OK
test-intersect-union...OK
test-intersect...OK
test-iter1...OK
test-iter3...OK
test-iter4...OK
test-lib-set...OK
test-lib...OK
test-local1...OK
test-local2...OK
test-ops1...OK
```

```
test-ops2...OK
test-perceptron...OK
test-print1...OK
test-print2...OK
test-set-assign...OK
test-set-bool...OK
test-set-call...OK
test-set...OK
test-set1...OK
test-slice...OK
test-std-set...OK
test-stdlib-append...OK
test-str-comp...OK
test-string...OK
test-unop1...OK
test-var1...OK
test-var2...OK
test-while...OK
test-while1...OK
test-while2...OK
test-while3...OK
fail-add1...OK
fail-append1...OK
fail-append2...OK
fail-assign1...OK
fail-assign2...OK
fail-card1...OK
fail-dead1...OK
fail-dead2...OK
fail-expr1...OK
fail-expr2...OK
fail-for3...OK
fail-func1...OK
fail-func2...OK
fail-func4...OK
fail-func6...OK
fail-func7...OK
fail-func8...OK
fail-func9...OK
fail-global2...OK
```

```
fail-if1...OK
fail-if2...OK
fail-if3...OK
fail-main...OK
fail-main2...OK
fail-main3...OK
fail-return1...OK
fail-return2...OK
fail-set1...OK
fail-set3...OK
fail-while1...OK
fail-while2...OK
```

7 Lessons Learned

One of the biggest things I learned was the need to consider the IR more when making design and implementation choices for the language, as the entire language is constrained by the IR in terms of these details. A lot of my implementation choices originally had fundamental problems (such as how I chose to initially determine and resolve the type for an empty set). By choosing to compile down to LLVM, I needed to make sure that all types were resolved during compile time and that all types were resolved in the correct way. For instance, a void array that gets used at some other point during the program and then gets declared back to void should retain the type it had during its use as this is the type LLVM needs to properly store the value it will contain in memory. Also, since pointers in C do not maintain a length, my design of a set had to be modified from pointers to structs that hold a pointer and an int value (representing the length). These choices revolved around thinking in terms of the capabilities of the IR.

From this project, I also learned the importance of having a test script. Whenever I modified an existing feature or something that existing features depend on, I would run the test script to make sure that I didn't break previous functionality. Some of the time, I did completely break functionality and, if I hadn't had this test script and hadn't been running it constantly, I would have lost the bug in hundreds of lines of code. Trying to find and fix it then would have been a nightmare. For instance, I used List.fold_left2 in

the codegen file on the bindings of a StringMap along with a list where the order of the two mattered. Everything broke after the next compile and it still took me a little to see that I was using StringMap.bindings when order mattered. Had I continued on it would have been much harder to fix.

8 Demos

All of the demo files are located in the *setc* directory under the *presentation* folder.

8.1 Demo 1

The first demo demonstrates basic SetC functionality. It is a bubble sort algorithm that simply passes in the set `a` from `main` to `bubble_sort`, which then sorts it using bubble sort. The printed output is the array in sorted order.

filename: `presentation/demo1.sc`

```
def main()
{
    a = [3,9,5,6,1,7,4,2,8];  /* initialization of a set */
    print(bubble_sort(a));
}


def bubble_sort(a) {
   swapped = true;
   while(swapped) { /* basic while loop */
        swapped = false;
         /* set theoretic iterator */
        (0<=j<(#a-1) | a[j] > a[j+1]) {
            swap(a, j);
            swapped = true;
        }
    }
    return a;
}

```

```
/* Function that swaps two adjacent
elements of a list */
def swap(a, j) {
    tmp = a[j];
    a[j] = a[j+1];
    a[j+1] = tmp;
}
```

```
$ ./setc.native < presentation/demo1.sc
>>
1
2
3
4
5
6
7
8
9
```

8.2 Demo 2

This demo is intended to demonstrate, more fully, function inferencing. Firstly, there is a recursive `gcd` algorithm that return the greatest common divisor of two integers, `a` and `b`. The second algorithm included is based off of Euler's phi function and returns a list of the elements up to `a` that are coprime to the integer `a`, that is, when their greatest common divisor is 1. The third algorithm takes the set that was produced by Euler's phi function of all coprime elements of `a` up to `a` and determines the coprimality of that set (whether all integers in the set are coprime to one another). The program outputs the set of all coprime numbers to 20 and whether that set is pairwise coprime. The function types, parameter types, and local variable types are all inferred.

filename: `presentation/demo2.sc`

```
def main() {
    print(coprime(phi(20)));
}

/* GCD algorithm */
```

```
def gcd(a, b) {
    if (a == 0) return b;
    return gcd(b%a, a);
}


/* Euler's phi function */
def phi (a) {
    l = [];
    (0<=i<a | gcd(i, a) == 1)
    l = l + [i];
    print(l);
    return l;
}

/* Coprimality in sets */
def coprime(s) {
    (0<=i<#s, (i+1)<=j<#s | gcd(s[j], s[i]) != 1)
            return false;
    return true;
}
```

```
$ ./setc.native < presentation/demo2.sc
>>
1
3
7
9
11
13
17
19
false
```

8.3 Demo 3

This demo represents the "coolest program" in the language. This algorithm is a machine learning classification algorithm: the perceptron learning algorithm. It operates on perfectly separable data.

It begins by opening an "input.txt" file to get the data (features and label values) of the examples. Each example is separated by a comma and each feature in an example is separated by a space. The program then splits this file by a comma to get the example and then by a space to get the features. Before appending the example, it appends a "1" that will simply act as the offset.

After the file has been transformed into an int array of the data, it runs the perceptron learning algorithm on this set. It beings with a set of zeroes representing initial beta values; these beta values serve to parametrize the hyperplane that will divide the positive and negative examples. Using the heuristic defined in function `f` to classify the example, it adjusts the weights using the algorithm defined in `adjust` if the classification was incorrect. After the algorithm converges (when the weights do not change anymore), these "optimal weights" are returned. The first of these determines the offset of the decision boundary and the rest determine its slope. The output is both printed out and outputted to a file, "results.txt".

filename: `presentation/perceptron.sc`

```
n = 4;

def main()
{
    a = open("input.txt", "r");
    f = get_input(a);
    close(a);
    w = PLA(f);
    output_results(w);
}

/* Writes the results to a file */
def output_results(w) {
    a = open("results.txt", "c");
    write(a, "Optimal Weights: ");
    (0<=i<#w) write(a, w[i]);
    close(a);
}

```

```
/* Reads and returns the data from file in a set */
def get_input(a) {
    file = read(a);
    b = split(file, ",");
    c = ["1"];
    d = [];
    (0<=i<#b) {
        e = split(b[i], " ");
        new = c + e;
        d = d + new;
    }
    f = [];
    (0<=i<#d) f = f + [str_to_int(d[i])];
    return f;
}

/* Performs PLA on the set a
   Returns optimal weights */
def PLA(a) {
    w = [0, 0, 0];
    change = true;
    while(change) {
        change = false;
        (0<=i<#a/n | f(a[n*i:(n*(i+1)-1)], w) * a[n*(i+1)-1
   ] <= 0) {
            change = true;
            adjust(a[n*i:(n*(i+1))], w);
        }
    }
    print("Optimal Weights");
    print(w);
    return w;
}

/* Heuristic function for PLA */
def f(data, w) {
    sum = 0;
    (0<=i<#data) sum = sum + w[i] * data[i];
    if (sum > 0) return 1;
    else {if (sum < 0) return -1; else return 0; }
```

```
}

/* Adjusts the weights */
def adjust(data, w) {
    (0<=i<#w) w[i] = w[i] + data[#data-1] * data[i];
}
```

Note: `run` is a script that simply runs the file through the compiler while also linking it with the `stdlib.o` file so that it has access to read, write, and split functionality defined in the `stdlib.c` file.

```
$ ./run presentation/perceptron.sc
>>
Optimal Weights
39
-5
-2
$ cat input.txt
>> 8 -11 1,7 7 -1,12 -20 1,14 -3 -1,12 8 -1,1 -12 1, 15 5
   -1, 7 -10 1, 10 4 -1, 6 2 1, 8 12 -1, 2 20 -1, 1 -12 1,
   9 8 -1, 3 3 1, 5 6 1, 1 11 1
$ cat results.txt
>>
Optimal Weights: 39
-5
-2
```

9 Appendix

9.1 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

```

```
all : setc.native strcmp.o stdlib.o

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

# "make clean" removes all generated files

.PHONY : clean
clean :
  ocamlbuild -clean
  rm -rf testall.log *.diff setc scanner.ml parser.ml parser.mli
  rm -rf strcmp
  rm -rf stdlib
  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 setc.cmx

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

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 $<

strcmp : strcmp.c
  cc -o strcmp -DBUILD_TEST strcmp.c
# Testing the "printbig" example
stdlib : stdlib.c
  cc -o stdlib -DBUILD_TEST stdlib.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
setc.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo
    ast.cmo
setc.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 =



FAILS =


```

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

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

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

9.2 setc.ml

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

type action = Ast | Sast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);      (* Print
   the AST only *)
                              ("-s", Sast);     (* Print
   the SAST only *)
                          ("-l", LLVM_IR);  (* Generate LLVM,
    don't check *)
                          ("-c", Compile) ] (* Generate,
   check LLVM IR *)
  else Compile in

  let file_to_string file =
    let array_string = ref [] in
      let ic = file in
          try
             while true do
                 array_string := List.append !array_string
   [input_line ic]
```

```
            done;
            String.concat "\n" !array_string
          with End_of_file -> close_in ic; String.concat "\
   n" !array_string

  in
  let in_file = open_in "stdlib.sc" in
  let string_in = file_to_string in_file in
  let other_file = file_to_string stdin in
  let str = String.concat "\n" [other_file; string_in] in


  let lexbuf = Lexing.from_string str  in
  let ast = Parser.program Scanner.token lexbuf  in
  let sast = Semant.check ast in
  match action with
    Ast -> print_string (Ast.string_of_program ast)
   | Sast -> print_string (Sast.sstring_of_program 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)
```

9.3 ast.ml

```
type op = Add | Sub | Mult | Div | Mod |  Equal | Neq
        | Less | Leq | Greater | Geq | And | Or | Carat |
   Qmark | Intersect | Union
type unop = Neg | Not | Card
type typ = Int | Void | String | Float | Bool
type datatype = Settype of typ | Datatype of typ



type expr =
        IntLit of int
        | FloatLit of float
        | StrLit of string
        | BoolLit of bool
```

```
14          | Set of expr list
15          | SetAccess of string * expr
16          | Id of string
17          | Unop of unop * expr
18          | Binop of expr * op * expr
19          | Call of string * expr list
20          | Slice of string * expr * expr
21          | Noexpr
22
23
24  and constraints = expr * op * string * op * expr
25
26  and stmt =
27          Block of stmt list
28          | Expr of expr
29          | If of expr * stmt * stmt
30          | While of expr * stmt
31          | Assign of string * expr
32          | SetElementAssign of string * expr * expr
33          | Iter of constraints list * expr * stmt
34          | Return of expr
35          | Break
36
37
38  type fdecl =  {
39      fname : string;
40      formals : string list;
41      body : stmt list;
42  }
43
44
45  type global =  string * expr
46
47  type program = global list * fdecl list
48
49
50  (* Pretty-printing functions (from MicroC) *)
51
52  let string_of_op = function
53      Add -> "+"
```

```
54    | Sub -> "-"
55    | Mult -> "*"
56    | Div -> "/"
57    | Mod -> "%"
58    | Equal -> "=="
59    | Neq -> "!="
60    | Less -> "<"
61    | Leq -> "<="
62    | Greater -> ">"
63    | Geq -> ">="
64    | And -> "&&"
65    | Or -> "||"
66    | Carat -> "^"
67    | Qmark -> "?"
68    | Intersect -> "&"
69    | Union -> "|"
70
71  let string_of_uop = function
72      Neg -> "-"
73    | Not -> "!"
74    | Card -> "#"
75
76
77  let rec string_of_expr = function
78      IntLit(l)               -> string_of_int l
79    | FloatLit(l)             -> string_of_float l
80    | StrLit(s)               -> s
81    | BoolLit(true)           -> "true"
82    | BoolLit(false)          -> "false"
83    | Id(s)                   -> s
84    | Binop(e1, o, e2)        -> string_of_expr e1 ^ " " ^
       string_of_op o ^ " " ^ string_of_expr e2
85    | Unop(o, e)              -> string_of_uop o ^
       string_of_expr e
86    | Call(f, el)             -> f ^ "(" ^ String.concat ", "
        (List.map string_of_expr el) ^ ")"
87    | Noexpr                  -> "noexpr"
88    | Set(l)                  -> "[" ^ String.concat " " (
       List.map string_of_expr l) ^ "]"
89    | SetAccess(s,e)          -> s ^ "[" ^ string_of_expr e ^
```

```
    "]"
  | Slice(s, e1, e2)        -> s ^ "[" ^ string_of_expr e1 ^ ":" ^ string_of_expr e2 ^ "]"


let string_of_constraints (e1, op1, s, op2, e2) =
     string_of_expr e1 ^ string_of_op op1 ^
     s ^ string_of_op op2 ^ string_of_expr e2

let rec string_of_stmt = function
    Block(stmts)                -> "Block{\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
  | While(e, s)                 -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Assign(st, e)               -> st ^ " = " ^ string_of_expr e ^ ";\n"
  | Iter(cl, e, s)              -> "(" ^ String.concat "" (List.map string_of_constraints cl) ^ "| "
                                    ^ string_of_expr e  ^ ")" ^ string_of_stmt s
  | SetElementAssign(s, e1, e2) -> s ^ "[" ^ string_of_expr e1 ^ "] = " ^ string_of_expr e2 ^ ";\n"
  | Break                       -> "break\n;"


let rec string_of_typ = function
    Datatype(Int) -> "int"
  | Datatype(String) -> "string"
```

```
117   | Datatype(Bool) -> "bool"
118   | Datatype(Void) -> "void"
119   | Datatype(Float) -> "float"
120   | Settype(s) -> string_of_typ (Datatype(s))
121
122
123 let string_of_vinit (s, e) = s ^ "=" ^ string_of_expr e ^ "
      ;\n"
124
125
126 let string_of_fdecl fdecl =
127   "def " ^ fdecl.fname ^ "(" ^ String.concat "," (List.map
      (fun x -> x) fdecl.formals) ^
128   ")\n{\n" ^
129   String.concat "" (List.map string_of_stmt fdecl.body) ^
130   "}\n"
131
132
133 let string_of_program (vars, funcs) =
134   String.concat "" (List.map string_of_vinit vars) ^ "\n" ^
135   String.concat "\n" (List.map string_of_fdecl funcs)
```

9.4 sast.ml

```
1  open Ast
2
3  type sexpr =
4          SIntLit of int * datatype
5          | SFloatLit of float * datatype
6          | SStrLit of string * datatype
7          | SBoolLit of bool * datatype
8          | SSet of sexpr list *  datatype
9          | SSetAccess of string * sexpr * datatype
10         | SId of string * datatype
11         | SUnop of unop * sexpr * datatype
12         | SBinop of sexpr * op * sexpr * datatype
13         | SCall of string * sexpr list * datatype
14         | SNoexpr
15
16
```

```
and sstmt =
        SBlock of sstmt list
        | SExpr of sexpr * datatype
        | SIf of sexpr * sstmt * sstmt
        | SWhile of sexpr * sstmt
        | SAssign of string * sexpr * datatype
        | SSetElementAssign of string * sexpr * sexpr *
   datatype
        | SIter of sconstraints *  sstmt
        | SReturn of sexpr * datatype
        | SBreak

and sconstraints = sstmt * sexpr * sstmt


type sfdecl = {
    styp : datatype;
    sfname : string;
    slocals : (string * datatype) list;
    sformals : (string * datatype) list;
    sbody : sstmt list;
}

type sglobal = string * sexpr * datatype

type sprogram = sglobal list * fdecl list

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

```
56    | Or -> "||"
57    | Carat -> "^"
58    | Qmark -> "?"
59    | Intersect -> "&"
60    | Union -> "|"
61
62  let sstring_of_uop = function
63      Neg -> "-"
64    | Not -> "!"
65    | Card -> "#"
66
67  let rec sstring_of_typ = function
68      Datatype(Int)       -> "int"
69    | Datatype(String)    -> "string"
70    | Datatype(Bool)      -> "bool"
71    | Datatype(Void)      -> "void"
72    | Datatype(Float)     -> "float"
73    | Settype(s)          -> "set(" ^ sstring_of_typ (
      Datatype(s)) ^ ")"
74
75
76  let rec sstring_of_expr = function
77      SIntLit(l,_t)           -> string_of_int l
78    | SFloatLit(l, _)         -> string_of_float l
79    | SStrLit(s, _)           -> s
80    | SBoolLit(true, _)       -> "true"
81    | SBoolLit(false, _)      -> "false"
82    | SId(s, _)               -> s
83    | SBinop(e1, o, e2, _)    ->
84        sstring_of_expr e1 ^ " " ^ sstring_of_op o ^ " " ^
      sstring_of_expr e2
85    | SUnop(o, e, _)          -> sstring_of_uop o ^
      sstring_of_expr e
86    | SCall(f, el, _)         -> f ^ "(" ^ String.concat ", "
       (List.map sstring_of_expr el) ^ ")"
87    | SNoexpr                 -> "noexpr"
88    | SSet(l, t)              -> "[" ^ String.concat " " (
      List.map sstring_of_expr l) ^ "]" ^ sstring_of_typ t
89    | SSetAccess(s,e,_)       -> s ^ "[" ^ sstring_of_expr e
      ^ "]"
```

```
90
91
92
93  let rec sstring_of_stmt = function
94      SBlock(stmts)                    -> "{\n" ^ String.
       concat "" (List.map sstring_of_stmt stmts) ^ "}\n"
95    | SExpr(expr, _)                   ->  sstring_of_expr
       expr ^ ";\n";
96    | SReturn(expr, _)                 -> "return " ^
       sstring_of_expr expr ^ ";\n";
97    | SIf(e, s, SBlock([]))            -> "if (" ^
       sstring_of_expr e ^ ")\n" ^ sstring_of_stmt s
98    | SIf(e, s1, s2)                   ->  "if (" ^
       sstring_of_expr e ^ ")\n" ^
99                                            sstring_of_stmt s1
       ^ "else\n" ^ sstring_of_stmt s2
100   | SWhile(e, s)                     -> "while (" ^
       sstring_of_expr e ^ ") " ^ sstring_of_stmt s
101   | SAssign(v, e, _)                 -> v ^ " = " ^
       sstring_of_expr e ^  ";\n"
102   | SIter(cl, body)                  -> "(" ^
       sstring_of_constraints cl ^ ") " ^ sstring_of_stmt body
103   | SBreak                           -> "break;\n"
104   | SSetElementAssign(s, e1, e2, _) -> s ^ "[" ^
       sstring_of_expr e1 ^ "] = " ^ sstring_of_expr e2 ^ ";\n"
105
106
107 and sstring_of_constraints (s1, e2, s3) =
108     sstring_of_stmt s1 ^ ", " ^ sstring_of_expr e2 ^ ", " ^
        sstring_of_stmt s3
109
110
111 let sstring_of_vinit (s, e, _) = s ^ " = " ^
       sstring_of_expr e ^ ";\n"
112
113
114 let sstring_of_fdecl fdecl =
115   sstring_of_typ fdecl.styp ^ " " ^
116   fdecl.sfname ^ "(" ^ String.concat ", " (List.map fst
       fdecl.sformals) ^
```

```
117   ") {\n" ^
118   String.concat "" (List.map sstring_of_stmt fdecl.sbody) ^
        "}" (*^
119   "}\n Locals:\n" ^ String.concat " " (List.map (fun (f, s)
        -> "name: " ^ f ^ " type: " ^ sstring_of_typ s ^"\n")
       fdecl.slocals) ^    "\nParams:\n" ^ String.concat " " (
       List.map (fun (f, s) -> "name: " ^ f ^ " type: " ^
       sstring_of_typ s ^"\n")  fdecl.sformals) *)
120
121
122 let sstring_of_program (vars, funcs) =
123   String.concat "" (List.map sstring_of_vinit vars) ^ "\n"
       ^
124   String.concat "\n" (List.map sstring_of_fdecl funcs)
```

9.5 scanner.mll

```
1  { open Parser }
2
3  let whitespace = [' ' '\t' '\r' '\n']
4  let digit = ['0'-'9']
5  let alpha = ['a'-'z' 'A'-'Z']
6  let ascii = [' '-'!' '#'-'[' ']'-'~']
7  let float = (digit+) ['.'] digit+
8  let int = digit+
9  let string = '"' ((ascii)* as s) '"'
10 let id = alpha (alpha | digit | '_')*
11
12
13 rule token = parse
14     whitespace { token lexbuf } (* Whitespace *)
15 | "/*"     { comment lexbuf }   (* Comments *)
16 | '('      { LPAREN }
17 | ')'      { RPAREN }
18 | '{'      { LBRACE }
19 | '}'      { RBRACE }
20 | '['      { LBRACK }
21 | ']'      { RBRACK }
22 | ';'      { SEMI }
23 | ','      { COMMA }
```

```
| ':'       { COLON }
| '|'       { STHAT }

(* Operators *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '%'       { MOD }

| '#'       { HASH }
| '?'       { QMARK }
| '&'       { UNION }

(* Equality Operators *)
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| "!"       { NOT }

(* Control *)
| "if"      { IF }
| "else"    { ELSE }
| "while"   { WHILE }
| "return"  { RETURN }
| "break"   { BREAK }

(* Keywords *)
| "def"     { DEF }
| "true"    { TRUE }
| "false"   { FALSE }

| int as lxm { INTLIT(int_of_string lxm) }
| float as lxm { FLOATLIT(float_of_string lxm) }
```

```
| string { STRLIT(s) }
| id as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.
    escaped char)) }

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

9.6 parser.mly

```
%{open Ast
module StringMap = Map.Make(String) %}

%token SEMI COLON LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
    COMMA
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN NOT UNION
%token EQ NEQ LT LEQ GT GEQ AND OR TRUE FALSE
%token RETURN DEF IF ELIF ELSE WHILE
%token BREAK
%token INT FLOAT VOID STRING BOOL
%token QMARK STHAT
%token HASH
%token ARRAY SET
%token <int> INTLIT
%token <float> FLOATLIT
%token <string> ID STRLIT
%token EOF

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

```
%left TIMES DIVIDE MOD UNION
%left QMARK
%left LBRACK
%right NOT NEG HASH
%nonassoc CONSTRAINT

%start program
%type <Ast.program> program

%%

program:
    decls EOF { $1 }

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

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


params_opt:
    /* nothing */ {[]}
    | param_list { List.rev $1 }

param_list:
    ID { [$1] }
    | param_list COMMA ID { $3 :: $1 }

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

stmt:
    | expr_stmt{ $1 }
```

```
68      | select_stmt { $1 }
69      | assign_stmt {  $1 }
70      | compound_stmt { $1 }
71      | iteration_stmt { $1 }
72      | jump_stmt { $1 }
73
74  expr_stmt:
75      expr SEMI { Expr $1 }
76
77  select_stmt:
78      IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
        Block([])) }
79      | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7)
        }
80
81  assign_stmt:
82      ID ASSIGN expr SEMI { Assign($1, $3) }
83      | ID LBRACK expr RBRACK ASSIGN expr SEMI {
        SetElementAssign($1, $3, $6)}
84
85
86  compound_stmt:
87      LBRACE stmt_list RBRACE { Block(List.rev $2) }
88
89  iteration_stmt:
90      WHILE LPAREN expr RPAREN stmt { While($3, $5) }
91      | LPAREN constraints_list expr_opt RPAREN stmt
92      { Iter( List.rev($2), $3, $5)  }
93
94  expr_opt:
95      /* nothing */ { Noexpr }
96      | STHAT expr { $2 }
97
98
99  constraints_list:
100     constraints { [$1] }
101     | constraints_list COMMA constraints { $3 :: $1 }
102
103
104 constraints:
```

```
105     expr LT ID LT expr { ($1, Less, $3, Less, $5) }
106   | expr LEQ ID LEQ expr { ($1, Leq, $3, Leq, $5)  }
107   | expr LT ID LEQ expr {($1, Less, $3, Leq, $5)  }
108   | expr LEQ ID LT expr {($1, Leq, $3, Less, $5)  }
109   | expr GT ID GT expr { ($1, Greater, $3, Greater, $5)  }
110   | expr GT ID GEQ expr {($1, Greater, $3, Geq, $5) }
111   | expr GEQ ID GEQ expr {($1, Geq, $3, Geq, $5) }
112   | expr GEQ ID GT expr {($1, Geq, $3, Greater, $5) }
113
114
115 jump_stmt:
116     | BREAK SEMI { Break }
117     | RETURN expr SEMI { Return($2) }
118
119 expr:
120     literals           { $1 }
121     | expr PLUS   expr { Binop($1, Add,   $3) }
122     | expr MINUS  expr { Binop($1, Sub,   $3) }
123     | expr TIMES  expr { Binop($1, Mult,  $3) }
124     | expr DIVIDE expr { Binop($1, Div,   $3) }
125     | expr MOD    expr { Binop($1, Mod,   $3) }
126     | expr EQ     expr { Binop($1, Equal, $3) }
127     | expr NEQ    expr { Binop($1, Neq,   $3) }
128     | expr LT     expr { Binop($1, Less,  $3) }
129     | expr LEQ    expr { Binop($1, Leq,   $3) }
130     | expr GT     expr { Binop($1, Greater, $3) }
131     | expr GEQ    expr { Binop($1, Geq,   $3) }
132     | expr AND    expr { Binop($1, And,   $3) }
133     | expr OR     expr { Binop($1, Or,    $3) }
134     | expr QMARK  expr { Binop($1, Qmark, $3) }
135     | expr UNION  expr { Binop($1, Union, $3) }
136     | MINUS expr %prec NEG { Unop(Neg, $2) }
137     | NOT expr         { Unop(Not, $2) }
138     | HASH expr        { Unop(Card, $2) }
139     | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
140     | LPAREN expr RPAREN { $2 }
141     | set { $1 }
142
143
144 literals:
```

```
145     INTLIT { IntLit($1) }
146     | FLOATLIT { FloatLit($1) }
147     | STRLIT { StrLit($1)}
148     | TRUE {BoolLit(true) }
149     | FALSE { BoolLit(false) }
150     | ID { Id($1) }
151
152
153 set:
154     LBRACK expr_list_opt RBRACK { Set($2) }
155     | ID LBRACK expr RBRACK { SetAccess($1, $3) }
156     | ID LBRACK expr COLON expr RBRACK { Slice($1, $3, $5)
        }
157
158
159 expr_list_opt:
160     /* nothing */ { [] }
161     | expr_list { List.rev $1 }
162
163 expr_list:
164     expr { [$1] }
165     | expr_list COMMA expr { $3 :: $1 }
166
167 actuals_opt:
168     /* nothing */ { [] }
169   | actuals_list  { List.rev $1 }
170
171 actuals_list:
172     expr                    { [$1] }
173   | actuals_list COMMA expr { $3 :: $1 }
174
175 vinit:
176    ID ASSIGN expr SEMI { ($1, $3) }
```

9.7 semant.ml

```
1 (* Semantic checking for the MicroC compiler *)
2
3 open Ast
4 open Sast
```

```
5  module E = Exceptions
6
7  module StringMap = Map.Make(String)
8
9
10 type env = {
11     env_fmap : fdecl StringMap.t;
12     env_fname : string;
13     env_return_type : Ast.datatype;
14     env_globals : Ast.datatype StringMap.t;
15     env_flocals : Ast.datatype StringMap.t;
16     env_in_while: bool;
17     env_set_return : bool;
18     env_sfmap : sfdecl StringMap.t
19 }
20
21 let rec expr_to_sexpr e env = match e with
22     IntLit(i)           -> (SIntLit(i, Datatype(Int)), env)
23   | FloatLit(b)         -> (SFloatLit(b, Datatype(Float)),
     env)
24   | StrLit(s)           -> (SStrLit(s, Datatype(String)),
     env)
25   | BoolLit(b)          -> (SBoolLit(b, Datatype(Bool)),
     env)
26   | Id(s)               -> (check_variable_access s env,
     env)
27   | Noexpr              -> (SNoexpr, env)
28   | Unop(op, e)         -> (check_unop op e env)
29   | Binop(e1, op, e2)   -> (check_binop e1 op e2 env)
30   | Call("print", e_l)  -> (check_print e_l env)
31   | Call("set", e_l)    -> (check_call_set e_l env)
32   (* Library functions *)
33   | Call("str", e_l)    -> (check_call_built "str" e_l env)
34   | Call("read", e_l)   -> (check_call_built "read" e_l env
     )
35   | Call("open", e_l)   -> (check_call_built "open" e_l env
     )
36   | Call("write", e_l)  -> (check_call_built "write" e_l
     env)
37   | Call("split", e_l)  -> (check_call_built "split" e_l
```

```
    env)
  | Call("close", e_l)  -> (check_call_built "close" e_l
    env)
  | Call(s, e_l)        -> (check_call s e_l env)
  | Set(e_l)            -> (check_set e_l env)
  | SetAccess(s, e)     -> (check_access s e env)
  | Slice(s, e1, e2)    -> (check_slice s e1 e2 env)


and check_slice s e1 e2 env =
    let ss = check_variable_access s env in
    let s_typ = try StringMap.find s env.env_flocals with
    Not_found -> StringMap.find s env.env_globals in
    let s_e_typ = match s_typ with Settype(a) -> a
            | _ -> (raise E.InvalidSetAccess) in
    let se1, env = expr_to_sexpr e1 env in
    let se2, env = expr_to_sexpr e2 env in
    let typ1 = sexpr_to_type se1 in
    let typ2 = sexpr_to_type se2 in
    if typ1 <> Datatype(Int) && typ2 <> Datatype(Int)
    then raise E.InvalidSliceOperation
    else match s_e_typ with
        Int -> (SCall("slicei", [ss; se1; se2], s_typ)),
    env
        | String -> (SCall("slices", [ss; se1; se2;], s_typ
    )), env
        | Float ->  (SCall("slicef", [ss; se1; se2;], s_typ
    )), env
        | Bool -> (SCall("sliceb", [ss; se1; se2;], s_typ))
    , env
        | _ -> raise E.InvalidSliceOperation


and check_access s e env =
    let _ = check_variable_access s env in
    let s_typ = try StringMap.find s env.env_flocals with
    Not_found -> StringMap.find s env.env_globals in
    let s_e_typ = match s_typ with Settype(a) -> a
            | _ -> (raise E.InvalidSetAccess) in
```

```
72      let se, env = expr_to_sexpr e env in
73      let typ = sexpr_to_type se in
74      if typ <> Datatype(Int) then raise E.InvalidSetAccess
75      else (SSetAccess(s, se, Datatype(s_e_typ)), env)
76
77
78  and check_set el env =
79      if (List.length el = 0) then SSet([], Settype(Void)),
        env
80      else
81          let se_first, env = expr_to_sexpr (List.hd el) env
        in
82          let e_typ = sexpr_to_type se_first in
83          let set_typ = match e_typ with
84              Datatype(t) -> t | _ -> (raise E.Invalid) in
85          let env_ref = ref (env) in
86          let se_l = List.rev(List.fold_left
87          (fun l e ->
88              let se, env = expr_to_sexpr e !env_ref in
89              env_ref := env;
90              let typ = sexpr_to_type se in
91              if (typ <> e_typ) then raise E.
        DifferentSetElementType
92              else (se :: l) ) [] (List.tl el)) in
93          let se_l_final = se_first :: se_l in
94          ((SSet(se_l_final, Settype(set_typ))), !env_ref)
95
96
97
98  and check_variable_access s env =
99      try let typ =  StringMap.find s env.env_flocals
100     in SId(s, typ) with Not_found ->
101         (try let typ = StringMap.find s
102     env.env_globals in SId(s, typ) with
103     Not_found -> (raise (E.UndefinedId s)))
104
105
106 and expr_list_to_sexpr_list e_l env =
107    let env_ref = ref(env) in
108    match e_l with
```

```
109     hd :: tl ->
110         let (se, env) = expr_to_sexpr hd !env_ref in
111         env_ref := env;
112         let (l, env) = expr_list_to_sexpr_list tl !env_ref
    in
113         env_ref := env;
114         (se :: l, !env_ref)
115   | [] -> ([], !env_ref)
116
117
118 and check_unop op e env =
119     let check_set_unop op = match op with
120         Card -> Datatype(Int)
121         | _  -> raise (E.InvalidUnaryOperation)
122     in
123     let check_bool_unop op = match op with
124         Not -> Datatype(Bool)
125         | _ -> raise (E.InvalidUnaryOperation)
126     in
127     let check_int_unop op = match op with
128         Neg -> Datatype(Int)
129         | _ -> raise (E.InvalidUnaryOperation)
130     in
131     let check_float_unop op = match op with
132         Neg -> Datatype(Float)
133         | _ -> raise (E.InvalidUnaryOperation)
134     in
135     let (se, env) = expr_to_sexpr e env in
136     let typ = sexpr_to_type se in
137     match typ with
138         Datatype(Int)         -> SUnop(op, se,
    check_int_unop op), env
139         | Datatype(Float)     -> SUnop(op, se,
    check_float_unop op), env
140         | Datatype(Bool)      -> SUnop(op, se,
    check_bool_unop op), env
141         | Settype(Void)    -> SIntLit(0, check_set_unop op),
     env
142         | Settype(_)          -> SUnop(op, se,
    check_set_unop op), env
```

```
143         | _ -> raise (E.InvalidUnaryOperation)
144
145
146 and check_binop e1 op e2 env =
147     let (se1, env) = expr_to_sexpr e1 env in
148     let (se2, env) = expr_to_sexpr e2 env in
149     let typ1 = sexpr_to_type se1 in
150     let typ2 = sexpr_to_type se2 in
151     match typ1, typ2 with Settype(t1), Settype(t2) ->
152             check_binop_set se1 op se2 t1 typ1 t2 env
153     | _ ->
154     match op with
155     Equal | Neq ->
156         if typ1 = typ2 || typ1 = Datatype(Void) || typ2 =
    Datatype(Void) then
157                     if typ1 = Datatype(String)
158                     then (SCall("strcomp", [se1;se2],
    Datatype(Bool)), env)
159                     else SBinop(se1, op, se2, Datatype(Bool
    )), env
160                   else raise(E.InvalidBinaryOperation)
161      | And | Or ->
162             if typ1 = Datatype(Bool) && typ2 = Datatype(
    Bool)
163             then SBinop(se1, op, se2, Datatype(Bool)), env
164             else raise(E.InvalidBinaryOperation)
165      | Less | Leq | Greater | Geq ->
166             if typ1 = typ2 &&
167             (typ1 = Datatype(Int) || typ1 = Datatype(Float)
    )
168                 then SBinop(se1, op, se2, Datatype(Bool)),
    env
169                 else raise(E.InvalidBinaryOperation)
170      | Add | Mult | Sub | Div | Mod -> if typ1 = typ2 &&
171             (typ1 = Datatype(Int) || typ1 = Datatype(Float)
    )
172                 then SBinop(se1, op, se2, typ1), env
173                 else raise(E.InvalidBinaryOperation)
174      | Qmark ->
175             let t = (match typ2 with
```

```
176                 Settype(t) -> Datatype(t)
177                 | _ ->  (raise E.InvalidBinaryOperation))
    in
178             if typ1 = t || t = Datatype(Void) then
179                 let new_se, env =
180                     match typ1 with
181                         Datatype(Int) -> expr_to_sexpr (
    Call("foundi", [e1; e2])) env
182                         | Datatype(Float) -> expr_to_sexpr
    (Call("foundf", [e1; e2])) env
183                         | Datatype(Bool) -> expr_to_sexpr (
    Call("foundb", [e1; e2])) env
184                         | Datatype(String) ->
    expr_to_sexpr (Call("founds", [e1; e2])) env
185                         | _ ->  raise E.
    InvalidBinaryOperation
186                         in
187                         new_se, env
188                     else
189                         (raise E.InvalidBinaryOperation)
190     | _ ->  raise(E.InvalidBinaryOperation)
191
192
193 and check_binop_set se1 op se2 t1 typ1 t2  env =
194     if not (t1 = Void || t2 = Void || t1 = t2) then raise E
    .InvalidBinaryOperation
195     else
196         (* Determine the type of the binop expr *)
197         let env, se1, se2, return, t1 =  match (t1, t2)
    with
198         (Void, a) ->
199             let se1, env = change_sexpr_type se1 (Settype(a
    )) env in env, se1, se2, a, a
200         | (a, Void) -> let se2, env = change_sexpr_type se2
     (Settype(a)) env in env, se1, se2, a, a
201         | (a, _) -> env, se1, se2, a, a
202         in
203         (* Convert binop to a call *)
204         match t1 with
205         Int -> (match op with
```

```
206             Add ->  SCall("append", [se1; se2], Settype(
    return)), env
207             | Sub -> SCall("diffi", [se1; se2], typ1), env
208             | Mult -> SCall("intersecti",[se1; se2], typ1),
     env
209             | Union -> SCall("unioni", [se1; se2], typ1),
    env
210             | _ -> raise E.InvalidBinaryOperation)
211         | Float -> (match op with
212             Add ->  SCall("append", [se1; se2], Settype(
    return)), env
213             | Sub -> SCall("difff", [se1; se2], typ1), env
214             | Mult -> SCall("intersectf",[se1; se2], typ1),
     env
215             | Union -> SCall("unionf", [se1; se2], typ1),
    env
216             | _ -> raise E.InvalidBinaryOperation)
217         | String -> (match op with
218             Add ->  SCall("append", [se1; se2], Settype(
    return)), env
219             | Sub -> SCall("diffs", [se1; se2], typ1), env
220             | Mult -> SCall("intersects",[se1; se2], typ1),
     env
221             | Union -> SCall("unions", [se1; se2], typ1),
    env
222             | _ -> raise E.InvalidBinaryOperation)
223         | Bool -> (match op with
224             Add ->  SCall("append", [se1; se2], Settype(
    return)), env
225             | Sub -> SCall("diffb", [se1; se2], typ1), env
226             | Mult -> SCall("intersectb",[se1; se2], typ1),
     env
227             | Union -> SCall("unionb", [se1; se2], typ1),
    env
228             | _ -> raise E.InvalidBinaryOperation)
229         | _ -> raise E.InvalidBinaryOperation
230
231
232 (* Check the set built in function *)
233 and check_call_set e env =
```

```
234     if List.length e <> 1 then
235     raise(E.WrongNumberOfArguments)
236     else
237         let (se, env) = expr_to_sexpr (List.hd e) env in
238         let typ = sexpr_to_type se in
239         let new_s, it = match typ with
240         | Settype(Int) -> "seti", Int
241         | Settype(Float) -> "setf", Float
242         | Settype(String) -> "sets", String
243         | Settype(Bool) -> "setb", Bool
244         | Settype(Void) -> "seti", Int
245         | _ -> (raise (E.IncorrectArgumentType("expected:
    settype",
246                             "found: " ^ string_of_typ typ)))
247         in
248         (SCall(new_s, [se], Settype(it)), env)
249
250
251 (* Check built in calls *)
252 and check_call_built str e env =
253     let fd = StringMap.find str env.env_fmap in
254     if List.length e <> List.length fd.formals then
255         raise(E.WrongNumberOfArguments)
256     else
257         let (se, env) = expr_list_to_sexpr_list e env in
258         (* Get the return type of the built in *)
259         let typ = match str with
260         "str" -> Datatype(String)
261         | "read" -> Datatype(String)
262         | "open" -> Datatype(Int)
263         | "write" -> Datatype(Void)
264         | "split" -> Settype(String)
265         | "str_to_int" -> Datatype(Int)
266         | "close" -> Datatype(Void)
267         | _ -> raise E.Invalid in
268         (SCall(str, se, typ), env)
269
270
271 and check_print e env =
272     if ((List.length e) <> 1) then raise (E.
```

```
   WrongNumberOfArguments)
273     else
274         let (se, env) = expr_to_sexpr (List.hd e) env in
275         let typ = sexpr_to_type se in
276         let new_s = match typ with
277         Datatype(Int) -> "print"
278         | Datatype(String) -> "prints"
279         | Datatype(Bool) -> "printb"
280         | Datatype(Float) -> "printf"
281         | Settype(t) -> (match t with
282                 | Int -> "printseti"
283                 | String -> "printsets"
284                 | Float -> "printsetf"
285                 | Bool -> "printsetb"
286                 | _ -> raise E.CannotPrintType)
287         | _ -> raise E.CannotPrintType
288         in
289         (SCall(new_s, [se], Datatype(Int)), env)
290
291
292 and check_call s e_l env =
293     (* Check if the function exists *)
294     if not (StringMap.mem s env.env_fmap)
295     then (print_string("fname: " ^ s); raise E.
   FunctionNotDefined)
296     else
297         (* Check that main wasn't called *)
298         if s = "main" then (raise E.CannotCallMain)
299         else
300             let env_ref = ref (env) in
301
302             (* List of semantically checked arguments  *)
303             let _ = List.rev(List.fold_left
304             (fun l expr ->
305                 let (se, env) = expr_to_sexpr expr env in
306                 env_ref := env;  se :: l) [] e_l )
307             in
308
309             (* List of the types of the arguments  *)
310             let arg_type_list = List.rev(List.fold_left
```

```
311             (fun l expr ->
312                 let (se, _) = expr_to_sexpr expr env in
313                 let typ = sexpr_to_type se in typ :: l) []
    e_l )
314              in
315
316             (* Check for correct number of arguments *)
317             let fd = StringMap.find s env.env_fmap in
318             if List.length e_l <> List.length fd.formals
    then
319             raise(E.WrongNumberOfArguments)
320             else
321                 (* if this function is called by itself *)
322                 if s = env.env_fname then
323                     let expr_list, env =
    expr_list_to_sexpr_list e_l env in
324                     ((SCall(s, expr_list, env.
    env_return_type), env))
325                 else
326                     (* If the function is called by another
     function
327                      * and the function has already  been
    called/ already defined(lib),
328                      * make sure types are correct *)
329                     if StringMap.mem s env.env_sfmap then
330                         let called_fdecl = StringMap.find s
     env.env_sfmap in
331                         List.iter2  (fun nt (_, ot) -> if
    nt <> ot && nt <> Settype(Void)
332                         then raise (E.IncorrectArgumentType
333                             ("expected: " ^ string_of_typ
    ot,
334                          "found: " ^ string_of_typ nt))
    else ())
335                         arg_type_list called_fdecl.sformals
    ;
336                         let expr_list, env =
    expr_list_to_sexpr_list e_l env in
337                         (SCall(s, expr_list,
338                             called_fdecl.styp), env)
```

```
339                     else
340                         (* Called for the first time:
    semantically check *)
341                         let expr_list, env =
    expr_list_to_sexpr_list e_l env in
342                         let new_env =
    convert_fdecl_to_sfdecl s arg_type_list env in
343                         let env =  {
344                             env_globals = env.env_globals;
345                             env_fmap = env.env_fmap;
346                             env_fname = env.env_fname;
347                             env_return_type = env.
    env_return_type;
348                             env_flocals = env.env_flocals;
349                             env_in_while = env.env_in_while
    ;
350                             env_set_return = env.
    env_set_return;
351                             env_sfmap = new_env.env_sfmap;
352                          } in
353                         let called_fdecl = StringMap.find s
     env.env_sfmap in
354                         (SCall(s, expr_list,
355                          called_fdecl.styp), env)
356
357
358
359 and sexpr_to_type sexpr = match sexpr with
360     SIntLit(_, typ)                  -> typ
361   | SFloatLit(_, typ)                -> typ
362   | SStrLit(_, typ)                  -> typ
363   | SBoolLit(_, typ)                 -> typ
364   | SId(_, typ)                      -> typ
365   | SBinop(_, _, _, typ)             -> typ
366   | SUnop(_, _, typ)                 -> typ
367   | SCall(_, _, typ)                 -> typ
368   | SSet(_, typ)                     -> typ
369   | SSetAccess(_,_,typ)              -> typ
370   | SNoexpr                          -> Datatype(Void)
371
```

```
372 and change_sexpr_type sexpr typ env = match sexpr with
373   SId(s,_)                          ->
374       (let flocals = StringMap.add s typ env.env_flocals
    in
375       let new_env = {
376            env_globals = env.env_globals;
377            env_fmap = env.env_fmap;
378            env_fname = env.env_fname;
379            env_return_type = env.env_return_type;
380            env_flocals = flocals;
381            env_in_while = env.env_in_while;
382            env_set_return = env.env_set_return;
383            env_sfmap = env.env_sfmap;
384        } in
385       (SId(s, typ), new_env))
386  | _ -> (sexpr, env)
387
388
389
390 and check_sblock sl env =
391     (* Make sure nothing follows a return *)
392     let rec check_block_return sl = match sl with
393         Return _ :: _ :: _ -> raise E.NothingAfterReturn
394       | Block sl :: ss -> check_block_return (sl @ ss)
395       | _ :: ss -> check_block_return ss
396       | [] -> ()
397     in check_block_return sl;
398
399     (* Check all statements within a block *)
400     match sl with
401     | [] ->   (SBlock([SExpr(SNoexpr, Datatype(Void))]),
    env)
402     | _ ->  let env_ref = ref(env) in
403       let block, env = (List.rev( List.fold_left
404       (fun l stmt ->
405           let new_stmt, env = convert_stmt_to_sstmt stmt
    !env_ref
406           in env_ref := env;
407           new_stmt :: l) [] sl), !env_ref)
408       in (SBlock(block), env)
```

```
409
410
411 and check_expr_stmt e env =
412     let se, env = expr_to_sexpr e env in
413     let typ = sexpr_to_type se in
414     (SExpr(se, typ), env)
415
416
417 and check_assign s e env =
418
419     let se, env = expr_to_sexpr e env in
420     let typ = sexpr_to_type se in
421
422     (* If the variable has already been declared, check new
        type *)
423     if StringMap.mem s env.env_flocals ||
424     StringMap.mem s env.env_globals then
425         let old_typ = try StringMap.find s env.env_flocals
426         with Not_found -> StringMap.find s env.env_globals
    in
427         (* Check set types *)
428         match old_typ, typ with
429         Settype(t1), Settype(t2) ->
430             (* If type mismatch *)
431             if not (t1 = Void || t2 = Void || t1 = t2) then
     (raise (E.AssignmentTypeMismatch
432                 ("expected set of: " ^ string_of_typ
    old_typ,
433                 "found set of : " ^ string_of_typ typ)))
434             else
435                 (* If new type was Void return the old_typ
    *)
436                 if t2 = Void then (SAssign(s, se, old_typ),
     env) else
437
438                 (* Add new settype to map *)
439                 let flocals = StringMap.add s typ env.
    env_flocals in
440                 let new_env = {
441                     env_globals = env.env_globals;
```

```
442                     env_fmap = env.env_fmap;
443                     env_fname = env.env_fname;
444                     env_return_type = env.env_return_type;
445                     env_flocals = flocals;
446                     env_in_while = env.env_in_while;
447                     env_set_return = env.env_set_return;
448                     env_sfmap = env.env_sfmap;
449                 }
450                 in
451                 (SAssign(s, se, typ), new_env)
452 
453         (* Check all other types *)
454         | _ ->
455             if (old_typ <> typ) then (raise
456             (E.AssignmentTypeMismatch("expected: " ^
    string_of_typ old_typ,
457             "found: " ^ string_of_typ typ)))
458         else (SAssign(s, se, typ), env)
459     else
460         (* Variable not declared yet, bind that type to the
     variable *)
461         let flocals = StringMap.add s typ env.env_flocals
    in
462         let new_env = {
463             env_globals = env.env_globals;
464             env_fmap = env.env_fmap;
465             env_fname = env.env_fname;
466             env_return_type = env.env_return_type;
467             env_flocals = flocals;
468             env_in_while = env.env_in_while;
469             env_set_return = env.env_set_return;
470             env_sfmap = env.env_sfmap;
471         }
472         in
473         (SAssign(s, se, typ), new_env)
474 
475 
476 and check_return e env =
477     let se, env  = expr_to_sexpr e env in
478     let typ = sexpr_to_type se in
```

```
479
480     (* If the return type has been set, check new return
        type *)
481     if env.env_set_return then
482         if env.env_return_type = typ
483         then (SReturn(se, typ), env)
484         else raise (E.ReturnTypeMismatch
485         ("expected: " ^ string_of_typ env.env_return_type,
486         "found: " ^ string_of_typ typ))
487     else
488         (* If no return type yet, make this the return type
        *)
489         let new_env = {
490             env_globals = env.env_globals;
491             env_fmap = env.env_fmap;
492             env_fname = env.env_fname;
493             env_return_type = typ;
494             env_flocals = env.env_flocals;
495             env_in_while = env.env_in_while;
496             env_set_return = true;
497             env_sfmap = env.env_sfmap;
498         }
499         in
500         (SReturn(se, typ), new_env)
501
502
503 and check_while e s env =
504     (* Create new env for in while *)
505     let old_env_in_while = env.env_in_while in
506     let env = {
507         env_globals = env.env_globals;
508         env_fmap = env.env_fmap;
509         env_fname = env.env_fname;
510         env_return_type = env.env_return_type;
511         env_flocals = env.env_flocals;
512         env_in_while = true;
513         env_set_return = env.env_set_return;
514         env_sfmap = env.env_sfmap;
515     }
516     in
```

```
517
518     (* Semantically check predicate and body *)
519     let (se, env) = expr_to_sexpr e env in
520     let typ = sexpr_to_type se in
521     let (while_body, env) = convert_stmt_to_sstmt s env in
522
523     (* Revert env variable *)
524     let env =  {
525         env_globals = env.env_globals;
526         env_fmap = env.env_fmap;
527         env_fname = env.env_fname;
528         env_return_type = env.env_return_type;
529         env_flocals = env.env_flocals;
530         env_in_while = old_env_in_while;
531         env_set_return = env.env_set_return;
532         env_sfmap = env.env_sfmap;
533     }
534     in
535
536     (* Make sure the predicate is of type Bool *)
537     if typ = Datatype(Bool) then
538         (SWhile(se, SBlock([while_body])), env)
539     else raise (E.InvalidWhileStatementType)
540
541
542
543 and check_if e s1 s2 env =
544     let (se, env) = expr_to_sexpr e env in
545     let typ = sexpr_to_type se in
546     let (if_body, env) =  convert_stmt_to_sstmt s1 env in
547     let (else_body, env) = convert_stmt_to_sstmt s2 env in
548     if typ = Datatype(Bool) then
549         (SIf(se, SBlock([if_body]), SBlock([else_body])),
    env)
550     else raise (E.InvalidIfStatementType)
551
552
553 and check_break env =
554     if env.env_in_while then
555         (SBreak, env)
```

```
556     else raise E.BreakOutsideOfLoop
557
558
559 (* Semantically check constraints and format them
560  * to be the structure of a for loop:
561      * initialization, predicate, increment *)
562 and check_constraints cl env =
563
564     (* check the bounds of the constraint *)
565     let check_expression e env =
566         let (se, env) = expr_to_sexpr e env in
567         let typ = sexpr_to_type se in
568         if typ = Datatype(Int) then (se, typ, env)
569         else raise E.InvalidIterStatementConstraintBounds
570     in
571     (* add the constraint variable to local variables *)
572     let add_iter_var s typ env =
573         let flocals = StringMap.add s typ env.env_flocals
    in
574         {
575             env_globals = env.env_globals;
576             env_fmap = env.env_fmap;
577             env_fname = env.env_fname;
578             env_return_type = env.env_return_type;
579             env_flocals = flocals;
580             env_in_while = env.env_in_while;
581             env_set_return = env.env_set_return;
582             env_sfmap = env.env_sfmap;
583         }
584     (* semantically check/change format of the constraint
    *)
585     in match cl with
586       (e1, Less, s, op, e2) ->
587           let se1, typ1, env = check_expression e1 env in
588           let env = add_iter_var s typ1 env in
589           let se2, typ2, env = check_expression e2 env in
590           ((SAssign(s, SBinop(se1, Add,
591                 SIntLit(1, Datatype(Int)), typ1), typ1),
592           SBinop(SId(s, typ1), op, se2, typ2),
593           SAssign(s, SBinop(SId(s, typ1), Add,
```

```
                SIntLit(1, Datatype(Int)), typ1), typ1)),
   env)
      |(e1, Leq, s, op, e2) ->
          let se1, typ1, env = check_expression e1 env in
          let env = add_iter_var s typ1 env in
          let se2, typ2, env = check_expression e2 env in
          ((SAssign(s, se1, typ1),
          SBinop(SId(s, typ1), op, se2, typ2),
          SAssign(s, SBinop(SId(s, typ1), Add,
                SIntLit(1, Datatype(Int)), typ1), typ1)),
   env)
      |(e1, Greater, s, op, e2) ->
          let se1, typ1, env = check_expression e1 env in
          let env = add_iter_var s typ1 env in
          let se2, typ2, env = check_expression e2 env in
          ((SAssign(s, SBinop(se1, Add,
                SIntLit(1, Datatype(Int)), typ1), typ2),
          SBinop(SId(s, typ1), op, se2, typ2),
          SAssign(s, SBinop(SId(s, typ1), Sub,
                SIntLit(1, Datatype(Int)), typ1), typ1)),
   env)
      |(e1, Geq, s, op, e2) ->
          let se1, typ1, env = check_expression e1 env in
          let env = add_iter_var s typ1 env in
          let se2, typ2, env = check_expression e2 env in
          ((SAssign(s, se1, typ2),
          SBinop(SId(s, typ1), op, se2, typ2),
          SAssign(s, SBinop(SId(s, typ1),
                Sub, SIntLit(1, Datatype(Int)), typ1), typ1
   )), env)
      |(_, _, _, _, _) -> raise E.InvalidIterStatement


and check_iter cl boole body env =
    (* Create new env in loop *)
    let old_env = env in
    let env = {
        env_globals = env.env_globals;
        env_fmap = env.env_fmap;
        env_fname = env.env_fname;
```

```
630         env_return_type = env.env_return_type;
631         env_flocals = env.env_flocals;
632         env_in_while = true;
633         env_set_return = env.env_set_return;
634         env_sfmap = env.env_sfmap;
635     }
636     in
637
638     (* Semantically check the constraints *)
639     let env_ref = ref(env) in
640     let constraints = List.rev(List.fold_left
641     (fun l c ->
642         let new_c, env = check_constraints c !env_ref
643         in env_ref := env;
644         (new_c :: l)) [] cl)
645     in
646
647     (* Semantically check the boolean statement *)
648     let (se, env) = expr_to_sexpr boole !env_ref  in
649     env_ref := env;
650     let typ = sexpr_to_type se in
651     let se, _ = match typ with
652         Datatype(Bool) -> se, typ
653         | Datatype(Void) -> SBoolLit(true, Datatype(Bool)),
     Datatype(Bool)
654         | _ -> raise E.InvalidIterStatementBoolExpression
655     in
656
657     (* Semantically check the body of the iter stmt *)
658     let (iter_body, env) = convert_stmt_to_sstmt body !
    env_ref in
659     env_ref := env;
660
661     (* Create a nested body for the siter stmt *)
662     let se_first_c = List.hd (List.rev constraints) in
663     let body = SIter(se_first_c, SIf(se, iter_body, SBlock
    ([]))) in
664     let body_ref = ref(body)
665     in
666     env_ref := {
```

```
667         env_globals = env.env_globals;
668         env_fmap = env.env_fmap;
669         env_fname = env.env_fname;
670         env_return_type = env.env_return_type;
671         env_flocals = env.env_flocals;
672         env_in_while = old_env.env_in_while;
673         env_set_return = env.env_set_return;
674         env_sfmap = env.env_sfmap;
675     };
676     let helper cl = match cl with
677         [] -> raise E.InvalidIterStatementConstraintBounds
678         | _ :: tl ->
679             (List.iter (fun cnstrnt ->
680                 let body = SIter(cnstrnt, SBlock([!
    body_ref])) in
681                 body_ref := body;) tl)
682     in helper (List.rev constraints); (!body_ref, !env_ref)
683
684
685 and check_set_assign s e1 e2 env =
686     (* Check if the variables has been created *)
687     let _ = check_variable_access s env in
688     let s_typ = try StringMap.find s env.env_flocals with
689     Not_found -> StringMap.find s env.env_globals in
690
691     (* Make sure it is of Settype *)
692     let s_e_typ = match s_typ with Settype(a) -> a
693         | _ -> raise E.InvalidSetAssign in
694     let se1, env = expr_to_sexpr e1 env in
695     let se2, env = expr_to_sexpr e2 env in
696     let typ1 = sexpr_to_type se1 in
697     let typ2 = sexpr_to_type se2 in
698
699     (* Make sure the access variable is an int and the
700      * element being assigned is of the set element type *)
701     if typ1 <> Datatype(Int) then raise E.InvalidSetAssign
702     else if typ2 <> Datatype(s_e_typ) then raise E.
    InvalidSetAssign
703     else SSetElementAssign(s, se1, se2, typ2), env
704
```

```
705
706
707 (* Semantically checks a statement *)
708 and convert_stmt_to_sstmt stmt env = match stmt with
709    Block sl              -> check_sblock sl env
710   | Expr e               -> check_expr_stmt e env
711   | Assign (s, e)        -> check_assign s e env
712   | Return e             -> check_return e env
713   | If(e, s1, s2)        -> check_if e s1 s2 env
714   | While(e, s)          -> check_while e s env
715   | Break                -> check_break env
716   | Iter(cl, e, s)       -> check_iter cl e s env
717   | SetElementAssign(s, e1, e2) -> check_set_assign s e1 e2
      env
718
719
720 (* Converts each fdecl to sfdecl--
721  * fname: function name
722  * arg_type_list: types of the parameters
723  * env: env variable *)
724 and convert_fdecl_to_sfdecl fname arg_type_list env =
725
726     let fdecl = StringMap.find fname env.env_fmap in
727
728     (* Make the parameters and their types local variables
    *)
729     let flocals = List.fold_left2 (fun m n t -> StringMap.
    add n t m)
730         StringMap.empty fdecl.formals arg_type_list
731     in
732
733     (* Starting env variable *)
734     let env =  {
735         env_globals = env.env_globals;
736         env_fmap = env.env_fmap;
737         env_fname = fdecl.fname;
738         env_return_type = Ast.Datatype(Int); (* placeholder
    until set *)
739         env_flocals = flocals;
740         env_in_while = false;
```

```
741         env_set_return = false;
742         env_sfmap = env.env_sfmap;
743     }
744     in
745
746     (* Semantically check all statements of the body *)
747     let (sstmts, env) = convert_stmt_to_sstmt (Block fdecl.
    body) env
748     in
749
750     (* Create semantically checked formals for fname *)
751     (* if a void type formal changed, change its type *)
752     report_duplicate(fdecl.formals);
753     let sformals = List.rev(List.fold_left2
754         (fun l name typ -> try (name, StringMap.find name
    env.env_flocals) :: l with Not_found-> (name, typ) :: l)
755         [] fdecl.formals arg_type_list)
756     in
757
758
759     (* Create semantically checked locals for fname *)
760     let formals_map = List.fold_left (fun m (n, t) ->
    StringMap.add n t m)
761     StringMap.empty sformals
762     in
763     let locals_map = List.fold_left (fun m (n,t)  ->
764         match t with Settype(Void) -> m | _ ->
765         if StringMap.mem n formals_map then StringMap.
    remove n m
766         else m) env.env_flocals (StringMap.bindings env.
    env_flocals)
767     in
768     let locals = StringMap.bindings locals_map
769     in
770     let locals = List.rev(List.fold_left (fun l (n, t) -> (
    n, t) :: l) [] locals) in
771
772     (* Create a semantically checked main function *)
773     let sfdecl = {
774         styp =  env.env_return_type;
```

```
775         sfname = fdecl.fname;
776         slocals = locals;
777         sformals = sformals;
778         sbody = match sstmts with SBlock(sl) ->  sl | _ ->
    [] ;
779     }
780     in
781
782     (* Return the env variable with this information *)
783     let env =  {
784         env_globals = env.env_globals;
785         env_fmap = env.env_fmap;
786         env_fname = env.env_fname;
787         env_return_type = env.env_return_type;
788         env_flocals = env.env_flocals;
789         env_in_while = env.env_in_while;
790         env_set_return = env.env_set_return;
791         env_sfmap = StringMap.add fname sfdecl env.
    env_sfmap;
792     }
793     in env
794
795
796 (* Reports duplicate variables *)
797 and report_duplicate list =
798     let rec helper = function
799         n1 :: n2 :: _ when n1 = n2 -> raise (E.
    DuplicateVariable n1)
800     | _ :: t -> helper t
801     | [] -> ()
802     in helper (List.sort compare list)
803
804 (* Converts lib to semantically checked lib *)
805 and convert_lib_fdecl_to_sfdecl env =
806
807     (* Library functions *)
808     let built_in_decls =
809         (StringMap.add "printb" [Datatype(Bool)]
810         (StringMap.add "printsetf" [Settype(Float)]
811         (StringMap.add "printsetb" [Settype(Bool)]
```

```
812         (StringMap.add "printseti" [Settype(Int)]
813         (StringMap.add "printsets" [Settype(String)]
814         (StringMap.add "sliceb" [Settype(Bool); Datatype(
    Int); Datatype(Int)]
815         (StringMap.add "slices" [Settype(String); Datatype(
    Int); Datatype(Int)]
816         (StringMap.add "slicef" [Settype(Float); Datatype(
    Int); Datatype(Int)]
817         (StringMap.add "slicei" [Settype(Int); Datatype(Int
    ); Datatype(Int)]
818         (StringMap.add "strcomp" [Datatype(String);
    Datatype(String)]
819         (StringMap.add "seti" [Settype(Int)]
820         (StringMap.add "setf" [Settype(Float)]
821         (StringMap.add "sets" [Settype(String)]
822         (StringMap.add "setb" [Settype(Bool)]
823         (StringMap.add "diffi" [Settype(Int); Settype(Int)]
824         (StringMap.add "difff" [Settype(Float); Settype(
    Float)]
825         (StringMap.add "diffs" [Settype(String); Settype(
    String)]
826         (StringMap.add "diffb" [Settype(Bool); Settype(Bool
    )]
827         (StringMap.add "unioni" [Settype(Int); Settype(Int)
    ]
828         (StringMap.add "unionf" [Settype(Float); Settype(
    Float)]
829         (StringMap.add "unions" [Settype(String); Settype(
    String)]
830         (StringMap.add "unionb" [Settype(Bool); Settype(
    Bool)]
831         (StringMap.add "intersecti" [Settype(Int); Settype(
    Int)]
832         (StringMap.add "intersectf" [Settype(Float);
    Settype(Float)]
833         (StringMap.add "intersects" [Settype(String);
    Settype(String)]
834         (StringMap.add "intersectb" [Settype(Bool); Settype
    (Bool)]
835         (StringMap.add "founds" [Datatype(String); Settype(
```

```
    String)]
836         (StringMap.add "foundi" [Datatype(Int); Settype(Int
    )]
837         (StringMap.add "foundf" [Datatype(Float); Settype(
    Float)]
838         (StringMap.add "foundb" [Datatype(Bool); Settype(
    Bool)]
839         (StringMap.add "appends" [Settype(String); Settype(
    String); Settype(String); Datatype(Int); Datatype(Int)]
840         (StringMap.add "appendf" [Settype(Float); Settype(
    Float); Settype(Float); Datatype(Int); Datatype(Int)]
841         (StringMap.add "appendb" [Settype(Bool); Settype(
    Bool); Settype(Bool); Datatype(Int); Datatype(Int)]
842         (StringMap.singleton "appendi" [Settype(Int);
    Settype(Int); Settype(Int); Datatype(Int); Datatype(Int)
    ]
843         ))))))))))))))))))))))))))))))))))
844     in
845
846     (* Update env variable through each function *)
847     let env_ref = ref(env) in
848     let _ = List.iter (fun (s, l) ->
849         let env = convert_fdecl_to_sfdecl s l !env_ref in
    env_ref := env)
850     (StringMap.bindings built_in_decls)
851     in !env_ref
852
853
854 (* Convert an ast tree to an sast tree *)
855 let convert_ast_to_sast globals functions fmap  =
856
857     (* Verify main function exists *)
858     let _ = try StringMap.find "main" fmap
859     with Not_found -> raise (E.MissingMainFunction) in
860
861     (* temp env var to check globals *)
862     let env =  {
863         env_globals = StringMap.empty;
864         env_fmap = fmap;
865         env_fname = "main";
```

```
866         env_return_type = Ast.Datatype(Int);
867         env_flocals = StringMap.empty;
868         env_in_while = false;
869         env_set_return = false;
870         env_sfmap = StringMap.empty;
871     }
872     in
873     (* Semantically check globals *)
874     (* TODO: check if globals are reassigned to an
    incompatible type *)
875     let env_ref = ref(env) in
876     let sglobals = List.rev (List.fold_left
877         (fun l (n, e) ->
878             let se, env = expr_to_sexpr e !env_ref
879             in let typ = sexpr_to_type se
880             in env_ref := env;
881             (n, se, typ) :: l) [] globals)
882     in
883
884     (* Create a map of the globals and add to env *)
885     let globals_map = List.fold_left (fun m (n, _, t) ->
886         StringMap.add n t m) StringMap.empty sglobals
887     in
888
889     (* Create env variable to start the semant check *)
890     let env =  {
891         env_globals = globals_map;
892         env_fmap = fmap;
893         env_fname = "main";
894         env_return_type = Ast.Datatype(Void);
895         env_flocals = StringMap.empty;
896         env_in_while = false;
897         env_set_return = false;
898         env_sfmap = StringMap.empty;
899     }
900     in
901
902     (* Check that they are no duplicate functions *)
903     report_duplicate (List.map (fun fd -> fd.fname)
    functions);
```

```
904
905      (* Convert all library functions *)
906     let env = convert_lib_fdecl_to_sfdecl env in
907
908     (* Convert fdecls to sfdecls through main *)
909     let env = convert_fdecl_to_sfdecl "main" [] env in
910
911     (* Convert all library functions *)
912     (*let env = convert_lib_fdecl_to_sfdecl env in*)
913
914     (* Return all semantically checked functions and
    globals *)
915     let sfdecls = List.rev(List.fold_left (fun l (_, sfd)
    -> sfd :: l)
916     [] (StringMap.bindings env.env_sfmap))
917     in (sglobals, sfdecls)
918
919
920 (* Add all functions declared (besides lib functions) to a
    function map *)
921 let build_fdecl_map functions =
922
923    (* Reserved functions *)
924    let built_in_decls =  StringMap.add "print"
925    { fname = "print"; formals = [("x")];
926    body =  []   } (StringMap.add "append"
927    { fname = "append"; formals = ["x"; "y"];
928    body =  []; }( StringMap.add "strcmp"
929    { fname = "strcmp"; formals = ["x"; "y"];
930    body = [] } (StringMap.add "set"
931    { fname = "set"; formals = ["x"];
932    body = [] }(StringMap.add "open"
933    { fname = "open"; formals = ["x"; "y"];
934    body = []; }  (StringMap.add "str"
935    { fname = "str"; formals = ["x"];
936    body = []; } (StringMap.add "pop"
937    { fname = "pop"; formals = ["x"];
938    body = []; } (StringMap.add "read"
939    { fname = "read"; formals = ["x"];
940    body = []; } (StringMap.add "write"
```

```
941     { fname = "write"; formals = ["x"; "y"];
942     body = []; } (StringMap.add "split"
943     { fname = "split"; formals = ["x"; "y"];
944     body = []; } (StringMap.add "str_to_int"
945     { fname = "str_to_int"; formals = ["x"];
946     body = []; } (StringMap.add "close"
947     { fname = "close"; formals = ["x"];
948     body = []; } (StringMap.add "write"
949     { fname = "write"; formals = ["x"; "y"];
950     body = []; }
951
952
953       (StringMap.singleton "prints"
954       { fname = "print"; formals = [("x")];
955         body =  []  }) ))))))))))))
956   in
957
958   (* Make sure none of the functions have a reserved name
959    * Note: lib functions are appended at end of file;
960    * error will come up as a duplicate function name *)
961   let check_functions m fdecl =
962       if StringMap.mem fdecl.fname m then
963           raise (E.DuplicateFunctionName fdecl.fname)
964       else if StringMap.mem fdecl.fname built_in_decls then
965           raise (E.FunctionNameReserved fdecl.fname)
966       else StringMap.add fdecl.fname fdecl m
967   in
968    List.fold_left (fun m fdecl -> check_functions m fdecl)
969   built_in_decls functions
970
971
972 let check (globals, functions) =
973     let fmap  = build_fdecl_map functions in
974     let sast = convert_ast_to_sast globals functions fmap
    in sast
```

9.8 codegen.ml

```
1 (* Code generation: translate takes a semantically checked
    AST and
```

```
2 produces LLVM IR
3
4 LLVM tutorial: Make sure to read the OCaml version of the
     tutorial
5
6 http://llvm.org/docs/tutorial/index.html
7
8 Detailed documentation on the OCaml LLVM library:
9
10 http://llvm.moe/
11 http://llvm.moe/ocaml/
12
13 *)
14
15 module L = Llvm
16 module A = Ast
17 module S = Sast
18 module E = Exceptions
19 module Semant = Semant
20
21 module StringMap = Map.Make(String)
22
23
24 let translate (globals, functions) =
25
26     let context = L.global_context () in
27     let the_module = L.create_module context "SetC"
28
29         and i32_t  = L.i32_type  context
30         and i8_t   = L.i8_type   context
31         and i1_t   = L.i1_type   context
32         and str_t  = L.pointer_type (L.i8_type context)
33         and float_t = L.double_type context
34         and void_t = L.void_type context in
35
36     let br_block    = ref (L.block_of_value (L.const_int
     i32_t 0)) in
37
38     let global_vars = ref (StringMap.empty) in
39     let local_vars = ref (StringMap.empty) in
```

```
40      let current_f = ref (List.hd functions) in
41      let set_lookup = ref (StringMap.empty) in
42
43      (* Pointer wrapper-- map of the named struct types
    represeting pointers. *)
44      let pointer_wrapper =
45          List.fold_left (fun m name -> StringMap.add name (L
    .named_struct_type context name) m)
46          StringMap.empty ["string"; "int"; "float"; "void";
    "bool"]
47      in
48      (* Set the struct body (fields) for each of the pointer
        struct types *)
49      List.iter2 (fun n l -> let t = StringMap.find n
    pointer_wrapper in
50      ignore(L.struct_set_body t (Array.of_list(l)) true))
51      ["float"; "int"; "string"; "void"; "bool"]
52      [[L.pointer_type float_t; i32_t; i32_t]; [L.
    pointer_type i32_t; i32_t; i32_t];
53      [L.pointer_type str_t; i32_t; i32_t];
54      [L.pointer_type void_t; i32_t; i32_t]; [L.pointer_type
    i1_t; i32_t; i32_t]];
55
56      (* Format strings for printing *)
57      let int_format_str builder = L.build_global_stringptr "
    %d\n" "fmt" builder
58      and str_format_str builder = L.build_global_stringptr "
    %s\n" "fmt" builder
59      and float_format_str builder = L.build_global_stringptr
        "%f\n" "fmt" builder in
60
61      (* Declare built in c functions (or c function wrappers
    ) *)
62      let printf_t = L.var_arg_function_type i32_t [| L.
    pointer_type i8_t |] in
63      let printf_func = L.declare_function "printf" printf_t
    the_module in
64
65      let strcmp_t = L.var_arg_function_type i32_t [| str_t;
    str_t |] in
```

```
66      let strcmp_func = L.declare_function "strcmp" strcmp_t
    the_module in
67
68      let strint_t = L.var_arg_function_type str_t [| str_t;
    L.pointer_type i32_t ; i32_t|] in
69      let strint_func = L.declare_function "strint" strint_t
    the_module in
70
71      let read_t = L.var_arg_function_type str_t [| i32_t |]
    in
72      let read_func = L.declare_function "read_sc" read_t
    the_module in
73
74      let writei_t =  L.var_arg_function_type i32_t [| i32_t;
     i32_t |] in
75      let writei_func = L.declare_function "writei_sc"
    writei_t the_module in
76
77      let writes_t =  L.var_arg_function_type i32_t [| i32_t;
     str_t |] in
78      let writes_func = L.declare_function "writes_sc"
    writes_t the_module in
79
80      let writef_t =  L.var_arg_function_type i32_t [| i32_t;
      float_t |] in
81      let writef_func = L.declare_function "writef_sc"
    writef_t the_module in
82
83      let close_t =  L.var_arg_function_type i32_t [| i32_t
    |] in
84      let close_func = L.declare_function "close_sc" close_t
    the_module in
85
86      let open_t =  L.var_arg_function_type i32_t [| str_t;
    str_t |] in
87      let open_func = L.declare_function "open_sc" open_t
    the_module in
88
89      let str_to_int_t =  L.var_arg_function_type i32_t [|
    str_t|] in
```

```
90      let str_to_int_func = L.declare_function "str_to_int"
        str_to_int_t the_module in
91
92      let split_t =  L.var_arg_function_type (L.pointer_type
        str_t) [| str_t; str_t |] in
93      let split_func = L.declare_function "split" split_t
        the_module in
94
95      let split_len_t =  L.var_arg_function_type i32_t [|
        str_t; str_t |] in
96      let split_len_func = L.declare_function "split_len"
        split_len_t the_module in
97
98
99      let rec string_of_typ datatype = match datatype with
100         A.Datatype(A.Int) -> "int"
101         | A.Datatype(A.String) -> "string"
102         | A.Datatype(A.Void) -> "void"
103         | A.Datatype(A.Bool) -> "bool"
104         | A.Datatype(A.Float) -> "float"
105         | A.Settype(t) -> string_of_typ (A.Datatype(t))
106     in
107
108     (* Gets the struct pointer *)
109     let lookup_struct typ =
110         let s = string_of_typ typ in
111         StringMap.find s pointer_wrapper in
112
113
114     (* Gets the llvm type of a datatype *)
115     let ltype_of_typ datatype = match datatype with
116         A.Datatype(A.Int) -> i32_t
117         | A.Datatype(A.String) -> str_t
118         | A.Datatype(A.Void) -> void_t
119         | A.Datatype(A.Bool) -> i1_t
120         | A.Datatype(A.Float) -> float_t
121         | A.Settype(t) -> L.pointer_type (lookup_struct (A.
        Datatype(t)))
122     in
123
```

```
124
125     (* StringMap of each function in the file (including
    lib functions) *)
126     let function_decls  =
127         let function_decl m fdecl =
128             let name = fdecl.S.sfname
129             and formal_types = Array.of_list (List.map
130             (fun (_, t) -> ltype_of_typ t) fdecl.S.sformals
    )
131             in
132             let ftype = L.function_type (ltype_of_typ fdecl
    .S.styp) formal_types in
133             StringMap.add name (L.define_function name
    ftype the_module, fdecl) m
134         in
135         List.fold_left function_decl StringMap.empty
    functions
136     in
137
138
139     let rec add_terminal builder f =
140         match L.block_terminator (L.insertion_block builder
    ) with
141       Some _ -> ()
142         | None -> ignore (f builder)
143
144
145     and expr builder = function
146         S.SIntLit (i, _) -> L.const_int i32_t i
147         | S.SStrLit (s, _) -> L.build_global_stringptr s "
    string" builder
148         | S.SBoolLit(b, _) -> L.const_int i1_t (if b then 1
     else 0)
149         | S.SFloatLit(f, _) -> L.const_float float_t f
150         | S.SNoexpr -> L.const_int i32_t 0
151         | S.SId (s, _) ->  L.build_load (lookup s) s
    builder
152         | S.SSet (el, t) ->
153                 let it = match t with A.Settype(it)-> it |
    _ -> raise E.Invalid in
```

```
154                 let struct_ptr = L.build_malloc (
    lookup_struct t) "set1" builder in
155                 let size = L.const_int i32_t ((List.length
    el) + 1) in
156                 let typ = L.pointer_type (ltype_of_typ (A.
    Datatype(it))) in
157                 let arr = L.build_array_malloc typ size "
    set2" builder in
158                 let arr = L.build_pointercast arr typ "set3
    " builder in
159                 let values = List.map (expr builder) el in
160                 let buildf i v = (let arr_ptr = L.build_gep
     arr
161                     [| (L.const_int i32_t (i + 1)) |] "set4
    " builder in
162                 ignore(L.build_store v arr_ptr builder);)
    in List.iteri buildf values;
163                 ignore(L.build_store arr (L.
    build_struct_gep struct_ptr 0 "set5" builder) builder);
164                 ignore(L.build_store (L.const_int i32_t (
    List.length el))
165                 (L.build_struct_gep struct_ptr 1 "set6"
    builder) builder);
166
167                 ignore(L.build_store (L.const_int i32_t 0)
    (L.build_struct_gep struct_ptr 2 "set7" builder) builder
    );
168                 struct_ptr
169
170         | S.SSetAccess (s, e, t) ->
171                 let idx = expr builder e in
172                 let idx = L.build_add idx (L.const_int
    i32_t 1) "access1" builder in
173                 let struct_ptr = expr builder (S.SId(s, t))
     in
174                 let arr = L.build_load (L.build_struct_gep
    struct_ptr 0 "access2" builder) "id1" builder in
175                 let res = L.build_gep arr [| idx |] "
    access3" builder in
176                 L.build_load res "access4" builder
```

```
177
178         | S.SBinop (e1, op, e2, _) ->
179           let e1' = expr builder e1
180           and e2' = expr builder e2 in
181             let typ = Semant.sexpr_to_type e1 in
182             (match typ with
183                 A.Datatype(A.Int) |  A.Datatype(A.Bool) ->
    (match op with
184               A.Add     ->   L.build_add
185               | A.Sub     -> L.build_sub
186               | A.Mult    -> L.build_mul
187                 | A.Div     -> L.build_sdiv
188               | A.And     -> L.build_and
189               | A.Or      -> L.build_or
190               | A.Equal   -> L.build_icmp L.Icmp.Eq
191               | A.Neq     -> L.build_icmp L.Icmp.Ne
192                 | A.Less    -> L.build_icmp L.Icmp.Slt
193               | A.Leq     -> L.build_icmp L.Icmp.Sle
194               | A.Greater -> L.build_icmp L.Icmp.Sgt
195               | A.Geq     -> L.build_icmp L.Icmp.Sge
196                 | A.Mod     -> L.build_srem
197                 | _         -> raise E.
    InvalidBinaryOperation
198             ) e1' e2' "tmp" builder
199               | A.Datatype(A.Float) -> (match op with
200             A.Add     ->   L.build_fadd
201             | A.Sub     -> L.build_fsub
202             | A.Mult    -> L.build_fmul
203               | A.Div     -> L.build_fdiv
204             | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
205             | A.Neq     -> L.build_fcmp L.Fcmp.One
206               | A.Less    -> L.build_fcmp L.Fcmp.Ult
207             | A.Leq     -> L.build_fcmp L.Fcmp.Ole
208             | A.Greater -> L.build_fcmp L.Fcmp.Ogt
209             | A.Geq     -> L.build_fcmp L.Fcmp.Oge
210               | A.Mod     -> L.build_frem
211               | _ -> raise E.InvalidBinaryOperation
212             ) e1' e2' "tmp" builder
213               | _ -> raise E.InvalidBinaryOperation)
214         | S.SUnop(op, e, _) ->
```

```
215           let e' = expr builder e in
216           (match op with
217               A.Neg       -> L.build_neg e' "tmp" builder
218             | A.Not       -> L.build_not e' "tmp" builder
219             | A.Card      ->
220                     let struct_ptr = expr builder e in
221                     L.build_load (L.build_struct_gep
    struct_ptr 1 "struct1" builder) "idl" builder)
222
223       | S.SCall("str_to_int", [e], _) ->
224             L.build_call str_to_int_func [| expr
    builder e|] "str_to_int" builder
225       | S.SCall("split", [e1; e2], _) ->
226             let struct_ptr = L.build_malloc (
    lookup_struct (A.Datatype(A.String))) "append3" builder
    in
227             let len = L.build_call split_len_func [|
    expr builder e1; expr builder e2 |] "split_len" builder
    in
228             let arr =  L.build_call split_func [| expr
    builder e1; expr builder e2 |] "split" builder in
229
230             ignore(L.build_store arr (L.
    build_struct_gep struct_ptr 0 "append8" builder) builder
    );
231             ignore(L.build_store len (L.
    build_struct_gep struct_ptr 1 "append9" builder) builder
    );
232             struct_ptr
233
234       | S.SCall("open", [e1; e2], _) ->
235             L.build_call open_func [| expr builder e1;
    expr builder e2 |] "open" builder
236
237       | S.SCall("read", [e1], _) ->
238             L.build_call read_func [| expr builder e1|]
     "read" builder
239       | S.SCall("write", [e1; e2], _) ->
240             let typ = Semant.sexpr_to_type e2 in
241             (match typ with
```

```
242                 A.Datatype(A.String) -> L.build_call
    writes_func [|expr builder e1; expr builder e2 |] "write
    " builder
243                 |A.Datatype(A.Int) -> L.build_call
    writei_func [|expr builder e1; expr builder e2 |] "write
    " builder
244                 | A.Datatype(A.Float) -> L.build_call
    writef_func [|expr builder e1; expr builder e2 |] "write
    " builder
245                 | _ -> raise E.Invalid)
246       | S.SCall("close", [e], _) ->
247                 L.build_call close_func [|expr builder e|]
    "close" builder
248
249       | S.SCall ("print", [e], _)  ->
250         L.build_call printf_func [| int_format_str
    builder ; (expr builder e) |]
251         "print" builder
252       | S.SCall ("prints", [e], _) ->
253           L.build_call printf_func [| str_format_str
    builder; (expr builder e) |]
254           "prints" builder
255       | S.SCall ("printf", [e], _) ->
256           L.build_call printf_func [| float_format_str
    builder; (expr builder e) |]
257           "printf" builder
258       | S.SCall ("strcmp", [e1; e2], _) ->
259           L.build_call strcmp_func [| expr builder e1;
    expr builder e2 |] "strcmp" builder
260       | S.SCall ("append", [e1; e2], t) ->
261                 let struct_ptr1 = expr builder e1 in
262                 let len1 = L.build_load(L.build_struct_gep
    struct_ptr1 1 "append1" builder) "tmp" builder  in
263                 let struct_ptr2 = expr builder e2 in
264                 let len2 = L.build_load(L.build_struct_gep
    struct_ptr2 1 "append2" builder) "tmp" builder  in
265
266                 let it = match t with A.Settype(it)-> it |
    _ -> raise E.Invalid in
267                 let struct_ptr = L.build_malloc (
```

```
    lookup_struct t) "append3" builder in
268             let s = L.build_add len1 len2 "append4"
    builder in
269             let size = L.build_add s (L.const_int i32_t
     1) "append5" builder in
270             let typ = L.pointer_type (ltype_of_typ (A.
    Datatype(it))) in
271             let arr = L.build_array_malloc typ size "
    append6" builder in
272             let arr = L.build_pointercast arr typ "
    append7" builder in
273
274             ignore(L.build_store arr (L.
    build_struct_gep struct_ptr 0 "append8" builder) builder
    );
275             ignore(L.build_store (s) (L.
    build_struct_gep struct_ptr 1 "append9" builder) builder
    );
276
277             ignore(L.build_store (L.const_int i32_t 0)
    (L.build_struct_gep struct_ptr 2 "append10" builder)
    builder);
278
279             let str = (match t with
280             A.Settype(A.Int) -> "appendi"
281             | A.Settype(A.Float) -> "appendf"
282             | A.Settype(A.String) -> "appends"
283             | A.Settype(A.Bool) -> "appendb"
284             | A.Settype(A.Void) -> raise E.Invalid
285             | _ -> raise E.Invalid) in
286
287             let (fdef, fdecl) = StringMap.find str
    function_decls in
288             let result = (match fdecl.S.styp with
289                             A.Datatype(A.Void) -> ""
290                             | _ -> str ^ "_result") in
291             L.build_call fdef [| struct_ptr1;
    struct_ptr2; struct_ptr; len1; len2 |] result builder
292
293       | S.SCall("set", act, t) ->
```

```
294                 let str = (match t with
295                 A.Settype(A.Int) -> "seti"
296                 | A.Settype(A.Float) -> "setf"
297                 | A.Settype(A.String) -> "sets"
298                 | A.Settype(A.Bool) -> "setb"
299                 | _ -> raise E.Invalid) in
300
301                 let (fdef, fdecl) = StringMap.find str
    function_decls in
302                 let actuals = List.rev (List.map (expr
    builder) (List.rev act)) in
303                 let result = (match fdecl.S.styp with
304                                     A.Datatype(A.Void) -> ""
305                                     | _ -> str ^ "_result") in
306                 L.build_call fdef (Array.of_list actuals)
    result builder
307       | S.SCall("str", [e], _) ->
308                 let struct_ptr = expr builder e in
309                 let int_ptr = L.build_load(L.
    build_struct_gep struct_ptr 0 "append1" builder) "tmp"
    builder  in
310                 let size = L.build_load(L.build_struct_gep
    struct_ptr 1 "append1" builder) "tmp" builder in
311                 let arr = L.build_array_malloc str_t size "
    set2" builder in
312                 let arr = L.build_pointercast arr str_t  "
    set3" builder in
313
314                 L.build_call strint_func [| arr; int_ptr ;
    size|] "strint" builder
315       | S.SCall("pop", [e], _) ->
316                 let struct_ptr = expr builder e in
317                 let arr_ptr = L.build_load(L.
    build_struct_gep struct_ptr 0 "append1" builder) "tmp"
    builder  in
318                 let size = L.build_load(L.build_struct_gep
    struct_ptr 1 "append1" builder) "tmp" builder in
319                 let new_size = L.build_sub (L.build_load(L.
    build_struct_gep struct_ptr 1 "append1" builder) "tmp"
    builder)
```

```
320                 (L.const_int i32_t 1) "sub" builder in
321                  ignore(L.build_store (new_size) (L.
    build_struct_gep struct_ptr 1 "append9" builder) builder
    );
322
323                 let res = L.build_gep arr_ptr [| size |] "
    pop3" builder in
324                 L.build_load res "pop4" builder
325
326
327       | S.SCall (f, act, _ ) ->
328           let (fdef, fdecl) = StringMap.find f
    function_decls in
329          let actuals = List.rev (List.map (expr builder) (
    List.rev act)) in
330          let result = (match fdecl.S.styp with
331                                  A.Datatype(A.Void) -> ""
332                                  | _ -> f ^ "_result") in
333           L.build_call fdef (Array.of_list actuals)
    result builder
334
335
336  and stmt builder =
337       let (the_function, _) = StringMap.find !current_f.S
    .sfname function_decls
338       in function
339
340       S.SBlock sl ->
341           List.fold_left stmt builder sl;
342       | S.SExpr (e, _) -> ignore (expr builder e);
    builder
343       | S.SAssign (s, e, _) ->
344                 let expr_t = Semant.sexpr_to_type e in
345                 (match expr_t with
346                 A.Settype(A.Void) ->
347                     if StringMap.find s !set_lookup = A.
    Void then (builder)
348                     else (
349                         let typ = StringMap.find s !
    set_lookup in
```

```
350                         let struct_ptr = L.build_malloc (
    lookup_struct (A.Datatype(typ))) "voidassign1" builder
    in
351                         let typ =  L.pointer_type (
    ltype_of_typ (A.Datatype(typ))) in
352                         let arr = L.const_pointer_null typ
    in
353                         ignore(L.build_store arr (L.
    build_struct_gep struct_ptr 0 "voidassign2" builder)
    builder);
354                         let size = L.const_int i32_t 0 in
355                         ignore(L.build_store size (L.
    build_struct_gep struct_ptr 1 "voidassign3" builder)
    builder);
356                         ignore(L.build_store struct_ptr (
    lookup s) builder); builder)
357
358               | _ ->
359                         (ignore(let e' = expr builder e in
360                         (L.build_store e' (lookup s)
    builder)); builder))
361
362       | S.SSetElementAssign(s, e1, e2, t) ->
363               let e2' = expr builder e2 in
364               let idx = expr builder e1 in
365               let idx = L.build_add idx (L.const_int
    i32_t 1) "setassign1" builder in
366               let struct_ptr = expr builder (S.SId(s, t))
     in
367               let arr = L.build_load(L.build_struct_gep
    struct_ptr 0 "setassign2" builder) "arr" builder  in
368               let res = L.build_gep arr [| idx |] "
    setassign3" builder in
369               ignore(L.build_store e2' res builder);
    builder
370
371       | S.SReturn (e, _) -> ignore (match !current_f.S.
    styp with
372         A.Datatype(A.Void) -> L.build_ret_void builder
373         | _ -> L.build_ret (expr builder e) builder);
```

```
   builder
374
375        | S.SIf (predicate, then_stmt, else_stmt) ->
376            let bool_val = expr builder predicate in
377          let merge_bb = L.append_block context "merge"
   the_function in
378
379          let then_bb = L.append_block context "then"
   the_function in
380          add_terminal (stmt (L.builder_at_end context
   then_bb) then_stmt)
381          (L.build_br merge_bb);
382
383          let else_bb = L.append_block context "else"
   the_function in
384           add_terminal (stmt (L.builder_at_end context
   else_bb) else_stmt)
385          (L.build_br merge_bb);
386
387          ignore (L.build_cond_br bool_val then_bb else_bb
   builder);
388          L.builder_at_end context merge_bb
389
390      | S.SWhile (predicate, body) ->
391
392          let pred_bb = L.append_block context "while"
   the_function in
393
394          let body_bb = L.append_block context "while_body"
    the_function in
395
396          let pred_builder = L.builder_at_end context
   pred_bb in
397          let bool_val = expr pred_builder predicate in
398
399          let merge_bb = L.append_block context "merge"
   the_function in
400
401            br_block  := merge_bb;
402
```

```
403             ignore(L.build_br pred_bb builder);
404
405             add_terminal (stmt (L.builder_at_end context
    body_bb) body)
406                 (L.build_br pred_bb);
407
408           ignore (L.build_cond_br bool_val body_bb merge_bb
     pred_builder);
409           L.builder_at_end context merge_bb
410
411       | S.SIter(constraints, body) ->
412             let (assign, predicate, increment) =
    constraints in
413             stmt builder (S.SBlock[assign;
414             S.SWhile(predicate, S.SBlock[body; increment])
    ])
415
416       | S.SBreak ->  ignore (L.build_br !br_block builder);
      builder
417
418
419     (* Lookup gives llvm for variable *)
420     and lookup n  = try StringMap.find n !local_vars
421         with Not_found ->   StringMap.find n !global_vars
422     in
423
424
425     (* Declare each global variable; remember its value in
    a map *)
426     let _global_vars =
427         let (f, _) = StringMap.find "main" function_decls
    in
428         let builder = L.builder_at_end context (L.
    entry_block f) in
429         let global_var m (n, e, _) =
430             let init = expr builder e
431             in StringMap.add n (L.define_global n init
    the_module) m in
432         List.fold_left global_var StringMap.empty globals
433     in global_vars := _global_vars;
```

```
434
435
436     let build_function_body fdecl =
437         let (the_function, _)  = StringMap.find fdecl.S.
    sfname function_decls
438         in let builder = L.builder_at_end context (L.
    entry_block the_function) in
439         current_f := fdecl;
440
441         (*Construct the function's "locals": formal
    arguments and locally
442         declared variables.  Allocate each on the stack,
    initialize their
443          value, if appropriate, and remember their values
    in the "locals" map *)
444
445         let _local_vars =
446             let add_formal m (n, t) p =
447                 L.set_value_name n p;
448
449                 let local = L.build_alloca (ltype_of_typ t)
     n builder
450                 in ignore (L.build_store p local builder);
    StringMap.add n local m
451             in
452             let add_local m (n, t) =
453                 (match t with
454                 | A.Settype(it) -> (ignore(set_lookup:=
    StringMap.add n it !set_lookup);
455                     let local_var = L.build_alloca (
    ltype_of_typ t) n builder
456                     in  StringMap.add n local_var m)
457                 | _ ->
458                         (let local_var = L.build_alloca (
    ltype_of_typ t) n builder
459                         in StringMap.add n local_var m)
460                 )
461             in
462             let formals = List.fold_left2 add_formal
    StringMap.empty
```

```
463                 fdecl.S.sformals (Array.to_list (L.params
    the_function))
464             in
465             List.fold_left add_local formals fdecl.S.
    slocals
466         in
467         local_vars := _local_vars;
468
469         (* Build the code for each statement in the
    function *)
470         let builder = stmt builder (S.SBlock fdecl.S.sbody)
     in
471
472         (* Add a return if the last block falls off the end
     *)
473         add_terminal builder (match fdecl.S.styp with
474             A.Datatype(A.Void) -> L.build_ret_void
475         | t -> L.build_ret (L.const_int (ltype_of_typ t) 0)
    )
476
477     in
478     List.iter build_function_body functions;
479     the_module
```

9.9 exceptions.ml

```
1  exception MissingEOF
2  exception Invalid
3  exception MissingMainFunction
4  exception CannotCallMain
5  exception WrongNumberOfArguments
6  exception IncorrectArgumentType of string * string
7  exception NothingAfterReturn
8  exception FunctionNotDefined
9  exception DuplicateFunctionName of string
10 exception FunctionNameReserved of string
11 exception CannotPrintType
12 exception InvalidUnaryOperation
13 exception InvalidSetAccess
14 exception InvalidBinaryOperation
```

```
exception DuplicateVariable of string
exception InvalidSliceOperation
exception AssignmentTypeMismatch of string * string
exception InvalidIterStatementConstraintBounds
exception InvalidIterStatementBoolExpression
exception InvalidIterStatement
exception NoConstraintBounds
exception DifferentSetElementType
exception InvalidIfStatementType
exception InvalidWhileStatementType
exception InvalidSetAssign
exception UndefinedId of string
exception ReturnTypeMismatch of string * string
exception BreakOutsideOfLoop
```

9.10 stdlib.sc

```
/* Functions for found "?" */
def foundi(a, b)
{
    if (#b == 0) return false;
    (0<=i<#b | b[i] == a) return true;
    return false;

}

def foundf(a,b)
{
    if (#b == 0) return false;
    (0<=i<#b | b[i] == a) return true;
    return false;
}

def founds(a,b) {
    (0<=i<#b | b[i] == a) return true;
    return false;
}

def foundb(a, b) {
    (0<=i<#b | b[i] == a) return true;
```

```
    return false;
}

/* Functions for settifying */
def seti(a)
{
    tmp = [];
    (0<=i<#a |  a[i]?tmp == false) tmp = tmp + [a[i]];
    return tmp;
}

def setf(a)
{
    tmp = [];
    (0<=i<#a |  a[i]?tmp == false) tmp = tmp + [a[i]];
    return tmp;
}

def sets(a)
{
    tmp = [];
    (0<=i<#a |  a[i]?tmp == false) tmp = tmp + [a[i]];
    return tmp;
}
def setb(a)
{
    tmp = [];
    (0<=i<#a |  a[i]?tmp == false) tmp = tmp + [a[i]];
    return tmp;
}

/* Functions for appending */
def appendi(arr1, arr2, arr, len1, len2)
{
    if (len1 == 0) return arr2;
    if (len2 == 0) return arr1;
    tmp1 = len1;
    tmp2 = len2;
    (0<=i<len1) arr[i] = arr1[i];
    (0<=i<len2) { j = i + len1; arr[j] = arr2[i]; }
```

```
    return arr;
}

def appendf(arr1, arr2, arr, len1, len2)
{
    if (len1 == 0) return arr2;
    if (len2 == 0) return arr1;
    tmp1 = len1;
    tmp2 = len2;
    (0<=i<len1) arr[i] = arr1[i];
    (0<=i<len2) { j = i + len1; arr[j] = arr2[i]; }
    return arr;
}

def appends(arr1, arr2, arr, len1, len2)
{
    if (len1 == 0) return arr2;
    if (len2 == 0) return arr1;
    tmp1 = len1;
    tmp2 = len2;
    (0<=i<len1) arr[i] = arr1[i];
    (0<=i<len2) { j = i + len1; arr[j] = arr2[i]; }
    return arr;
}

def appendb(arr1, arr2, arr, len1, len2)
{
    if (len1 == 0) return arr2;
    if (len2 == 0) return arr1;
    tmp1 = len1;
    tmp2 = len2;
    (0<=i<len1) arr[i] = arr1[i];
    (0<=i<len2) { j = i + len1; arr[j] = arr2[i]; }
    return arr;
}


/* Functions for intersection */
def intersecti(a, b)
{
```

```
104     tmp = [];
105     (0<=i<#a, 0<=j<#b | a[i] == b[j] && !(a[i]?tmp))
106     tmp = tmp + [a[i]];
107     return tmp;
108 }
109
110 def intersectf(a, b) {
111     tmp = [];
112     (0<=i<#a, 0<=j<#b | a[i] == b[j] && !(a[i]?tmp))
113     tmp = tmp + [a[i]];
114     return tmp;
115 }
116
117 def intersects(a, b) {
118     tmp = [];
119     (0<=i<#a, 0<=j<#b | a[i] == b[j] && !(a[i]?tmp))
120     tmp = tmp + [a[i]];
121     return tmp;
122 }
123 def intersectb(a, b) {
124     tmp = [];
125     (0<=i<#a, 0<=j<#b | a[i] == b[j] && !(a[i]?tmp))
126      tmp = tmp + [a[i]];
127     return tmp;
128 }
129
130
131
132 /* Functions for union */
133 def unioni(a, b) {
134     return set(a+b);
135 }
136
137 def unionf(a, b) {
138     return set(a+b);
139 }
140
141 def unions(a, b) {
142     return set(a+b);
143 }
```

```
144 def unionb(a, b) {
145     return set(a+b);
146 }
147
148
149
150 /* Functions for difference */
151 def diffi(a, b) {
152     tmp = [];
153     (0<=i<#a | !(a[i]?b)) tmp = tmp + [a[i]];
154     return tmp;
155 }
156
157 def difff(a, b) {
158     tmp = [];
159     (0<=i<#a | !(a[i]?b)) tmp = tmp + [a[i]];
160     return tmp;
161 }
162
163 def diffs(a, b) {
164     tmp = [];
165     (0<=i<#a | !(a[i]?b)) tmp = tmp + [a[i]];
166     return tmp;
167 }
168 def diffb(a, b) {
169     tmp = [];
170     (0<=i<#a | !(a[i]?b)) tmp = tmp + [a[i]];
171     return tmp;
172 }
173
174 /* Function for string comparison */
175 def strcomp(a, b) {
176     c = strcmp(a, b);
177     if (c == 0) return true;
178     else return false;
179 }
180
181 /* Functions for slicing */
182 def slicei(a, b, c) {
183     d = [];
```

```
184     (b<=i<c) d = d + [a[i]];
185     return d;
186 }
187
188 def slicef(a, b, c) {
189     d = [];
190     (b<=i<c) d = d + [a[i]];
191     return d;
192 }
193
194 def slices(a, b, c) {
195     d = [];
196     (b<=i<c) d = d + [a[i]];
197     return d;
198 }
199
200 def sliceb(a, b, c) {
201     d = [];
202     (b<=i<c) d = d + [a[i]];
203     return d;
204 }
205
206 /* Functions for printing sets */
207 def printsets(a) {
208     (0<=i<#a) print(a[i]);
209 }
210
211 def printseti(a) {
212     (0<=i<#a) print(a[i]);
213 }
214
215 def printsetf(a) {
216     (0<=i<#a) print(a[i]);
217 }
218
219 def printsetb(a) {
220     (0<=i<#a) print(a[i]);
221 }
222
223 /* Function for printing bool */
```

```
def printb(a) {
    if (a==false) print("false");
    else print("true");
}
```

9.11 stdlib.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<unistd.h>

int split_len(char* str, char* delimiter) {
    int i = 0;
    int number = 0;
    while (i < strlen(str) && str[i] == delimiter[0]) i++;
    while (i < strlen(str) && i+1 < strlen(str)){
        if (str[i] == delimiter[0] && str[i+1] != delimiter
   [0]) {
            number++;
        }
        i++;
    }
    return number + 1;
}

char** split(char* str, char* delimiter) {
    int i;
    int number = 0;
    char ** new;
    char* token;
    char * extra;
    extra = malloc(strlen(str) + 1 * sizeof(char));

    strncpy(extra, str, strlen(str));
    extra[strlen(str)] = 0;

    number = split_len(str, delimiter);

```

```
33      new = (char **)malloc((number + 1)* sizeof(char *));
34
35      token = strtok(extra, delimiter);
36      new[0] = "hi";
37      i = 1;
38      while(token!=NULL) {
39          new[i] = token;
40          token = strtok(NULL, delimiter);
41          i++;
42      }
43
44      return new;
45  }
46
47
48  int str_to_int(char* b) {
49      return atoi(b);
50  }
51
52  int open_sc(char* file, char* method) {
53      int i = 0;
54      int fd;
55      switch(method[0]) {
56          case 'r':
57             i =  O_RDONLY;
58             break;
59          case 'w':
60             i = O_WRONLY;
61             break;
62          case 'a':
63             i = O_APPEND;
64             break;
65          case 'c':
66             i = O_WRONLY | O_CREAT;
67             break;
68          default:
69             i = O_RDONLY;
70             break;
71      }
72      fd = open(file, i, 0640);
```

```
73
74      return fd;
75  }
76
77  char* read_sc(int fd) {
78
79      char * buffer = NULL;
80      int pos;
81      size_t size;
82
83      pos = lseek(fd, (size_t)0, SEEK_CUR);
84      size = lseek(fd, (size_t)0, SEEK_END);
85      lseek(fd, pos, SEEK_SET);
86
87      buffer = malloc((size + 1) * sizeof(*buffer));
88
89      read(fd, buffer, size);
90      buffer[size] = '\0';
91      return buffer;
92  }
93
94  int writes_sc(int fd, char* ptr) {
95      int len = strlen(ptr);
96      return write(fd, ptr, len);
97  }
98
99  int writei_sc(int fd, int ptr) {
100     char tmp[12];
101     memset(tmp, 0, sizeof(tmp));
102     snprintf(tmp, sizeof(int*), "%d\n", ptr);
103     return write(fd, tmp, strlen(tmp));
104 }
105
106
107 int writef_sc(int fd, double ptr) {
108     char tmp[12];
109     memset(tmp, 0, sizeof(tmp));
110     snprintf(tmp, sizeof(double*), "%f\n", ptr);
111     return write(fd, tmp, sizeof(double));
112 }
```

```
113
114 int close_sc(int fd) {
115     return close(fd);
116 }
117
118
119 char* strint(char* char_arr, int* int_arr, int len)
120 {
121     int i;
122     for (i=0; i<len; i++) {
123         char_arr[i] = (char) (int_arr[i+1] + '0');
124     }
125     return char_arr;
126 }
```

9.12 testall.sh

```
1  #!/bin/sh
2
3  # Regression testing script for MicroC
4  # Step through a list of files
5  #  Compile, run, and check the output of each expected-to-
       work test
6  #  Compile and check the error of each expected-to-fail
       test
7
8  # Path to the LLVM interpreter
9  LLI="lli"
10 #LLI="/usr/local/opt/llvm/bin/lli"
11
12 # Path to the LLVM compiler
13 LLC="llc"
14
15 # Path to the C compiler
16 CC="cc"
17
18 # Path to the microc compiler.  Usually "./microc.native"
19 # Try "_build/microc.native" if ocamlbuild was unable to
       create a symbolic link.
20 SETC="./setc.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] [.sc 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>
```

```
60 # Report the command, run it, and report any errors
61 Run() {
62     echo $* 1>&2
63     eval $* || {
64   SignalError "$1 failed on $*"
65   return 1
66     }
67 }
68
69 # RunFail <args>
70 # Report the command, run it, and expect an error
71 RunFail() {
72     echo $* 1>&2
73     eval $* && {
74   SignalError "failed: $* did not report an error"
75   return 1
76     }
77     return 0
78 }
79
80 Check() {
81     error=0
82     basename=`echo $1 | sed 's/.*\\///
83                               s/.sc//'`
84     reffile=`echo $1 | sed 's/.sc$//'`
85     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
86
87     echo -n "$basename..."
88
89     echo 1>&2
90     echo "###### Testing $basename" 1>&2
91
92     generatedfiles=""
93
94     generatedfiles="$generatedfiles ${basename}.ll ${
   basename}.s ${basename}.exe ${basename}.out" &&
95     Run "$SETC" "<" $1 ">" "${basename}.ll" &&
96     Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
97     Run "$CC" "-o" "${basename}.exe" "${basename}.s" "
   stdlib.o" &&
```

```
98      Run "./${basename}.exe" > "${basename}.out" &&
99      Compare ${basename}.out ${reffile}.out ${basename}.diff
100
101     # Report the status and clean up the generated files
102
103     if [ $error -eq 0 ] ; then
104   if [ $keep -eq 0 ] ; then
105       rm -f $generatedfiles
106   fi
107   echo "OK"
108   echo "###### SUCCESS" 1>&2
109     else
110   echo "###### FAILED" 1>&2
111   globalerror=$error
112     fi
113 }
114
115 CheckFail() {
116     error=0
117     basename=`echo $1 | sed 's/.*\\///
118                               s/.sc//'`
119     reffile=`echo $1 | sed 's/.sc$//'`
120     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
121
122     echo -n "$basename..."
123
124     echo 1>&2
125     echo "###### Testing $basename" 1>&2
126
127     generatedfiles=""
128
129     generatedfiles="$generatedfiles ${basename}.err ${
    basename}.diff" &&
130     RunFail "$SETC" "<" $1 "2>" "${basename}.err" ">>"
    $globallog &&
131     Compare ${basename}.err ${reffile}.err ${basename}.diff
132
133     # Report the status and clean up the generated files
134
135     if [ $error -eq 0 ] ; then
```

```
136   if [ $keep -eq 0 ] ; then
137       rm -f $generatedfiles
138   fi
139   echo "OK"
140   echo "###### SUCCESS" 1>&2
141     else
142   echo "###### FAILED" 1>&2
143   globalerror=$error
144     fi
145 }
146 
147 while getopts kdpsh c; do
148     case $c in
149   k) # Keep intermediate files
150       keep=1
151       ;;
152   h) # Help
153       Usage
154       ;;
155     esac
156 done
157 
158 shift `expr $OPTIND - 1`
159 
160 LLIFail() {
161   echo "Could not find the LLVM interpreter \"$LLI\"."
162   echo "Check your LLVM installation and/or modify the LLI
     variable in testall.sh"
163   exit 1
164 }
165 
166 which "$LLI" >> $globallog || LLIFail
167 
168 if [ $# -ge 1 ]
169 then
170     files=$@
171 else
172     files="tests/test-*.sc tests/fail-*.sc"
173 fi
174 
```

```
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
```

9.13 Tests

```
def main(){
a = 1;
b = true;
c = a + b;
print(a);
}

def foo(a, b){
c = a + b;
return (#c);
}

def main(){
a = [0, 1];
b = [true, false];
c = foo(a, b);
print(c);
}
def foo(){
a = [1];
```

```
b = ["a"];
c = a + b;
print(c);
}

def main(){
foo();
}
def main()
{
  i = 42;
  i = 10;
  b = true;
  b = false;
  i = false; /* Fail: assigning a bool to an integer */
}
def main()
{
  i = 1;
  b = false;

  b = 48; /* Fail: assigning an integer to a bool */
}
def main(){
    a = [1,2,3,4,5];
    b = ["a", "b", "c"];
    c = 1;
    #a;
    #b;
    #c;
}
def main()
{
  i = 15;
  return i;
  i = 32; /* Error: code after a return */
}
def main()
{
  i = 5;
```

```
61
62    {
63      i = 15;
64      return i;
65    }
66    i = 32; /* Error: code after a return */
67  }
68  a = 1;
69  b = true;
70
71  def foo(c, d)
72  {
73    dd = 11;
74    e = false;
75    a + c;
76    c - a;
77    a * 3;
78    c / 2;
79    d + a; /* Error: bool + int */
80  }
81
82  def main()
83  {
84    foo(2, false);
85    return 0;
86  }
87  a = 1;
88  b = 2;
89
90  def foo(c, d)
91  {
92    d = 3;
93    e = true;
94    b + e; /* Error: bool + int */
95  }
96
97  def main()
98  {
99    foo(1,2);
100   return 0;
```

```
}
def bar(a){
  return a;
}

def foo(){
  (0 < bar(3) < 3 | true)  a= 1;
  return 1;
}

def main(){
  print(foo());
  return 0;
}
def foo() {}

def bar() {}

def baz() {}

def bar() {} /* Error: duplicate function bar */

def main()
{
  return 0;
}
def foo(a, b, c) { }

def bar(a, b, a) {} /* Error: duplicate formal a in bar */

def main()
{
  bar(1, 2, 3);
  return 0;
}
def foo() {}

def bar() {}

def print() {} /* Should not be able to define print */
```

```
141
142 def baz() {}
143
144 def main()
145 {
146   return 0;
147 }
148 def foo(a, b)
149 {
150 }
151
152 def main()
153 {
154   foo(42, true);
155   foo(42); /* Wrong number of arguments */
156 }
157 def foo(a, b)
158 {
159 }
160
161 def main()
162 {
163   foo(42, true);
164   foo(42, true, false); /* Wrong number of arguments */
165 }
166 def foo(a, b)
167 {
168 }
169
170 def bar()
171 {
172 }
173
174 def main()
175 {
176   foo(42, true);
177   foo(42, bar()); /* int and void, not int and bool */
178 }
179 def foo(a, b)
180 {
```

```
}

def main()
{
  foo(42, true);
  foo(42, 42); /* Fail: int, not bool */
}
b = 1;
c = true;
a = 5;
b = false; /* Duplicate global variable */

def main()
{
  return 0;
}
def main()
{
  if (true) {}
  if (false) {} else {}
  if (42) {} /* Error: non-bool predicate */
}
def main()
{
  if (true) {
    foo; /* Error: undeclared variable */
  }
}
def main()
{
  if (true) {
    42;
  } else {
    bar; /* Error: undeclared variable */
  }
}
a = 1;

def main(){
  if(a != 3){
```

```
    a = a + 1;
  }
main();
return 0;
}
def main(){
  foo(1);
return 0;
}

def foo(a){
  main();
  return 0;
}
def func(){}
def main()
{
  if(true) return 5;
  return true; /* error with different return types*/
}
def foo()
{
  if (true) return 42;
  else return "hi";
    /* different return types, void type */
}

def main()
{
  foo();
  return 42;
}
def main(){
a = [1, 2, "a", "b"]; /* fail due to multiple types */
print(a);
}
def foo(a, b, c){
set3 = a + b;
set4 = set3 + c; /* should fail, set + bool*/
}
```

```
261
262 def main(){
263 set1 = ["a", "b", "c", "d"];
264 set2 = [1, 2, 3, 4, 5];
265 c = true;
266 foo(set1, set2, c);
267 return 0;
268 }
269 def main()
270 {
271   i = 1;
272
273   while (i != 5) {
274     i = i + 1;
275   }
276
277   while (42) { /* Should be boolean */
278     i = i + 1;
279   }
280
281 }
282 def main()
283 {
284   i = 1;
285
286   while (i != 5) {
287     i = i + 1;
288   }
289
290   while (true) {
291     foo(); /* foo undefined */
292   }
293
294 }
295 def add(x, y)
296 {
297   return x + y;
298 }
299
300 def main()
```

```
{
  print(add(17, 25) );
  return 0;
}
def main() {
a = 1;
c = 3 + a;
print(c);
d = 3;
return 3;
}
def main() {
b = [];
a= [1,2];
b = b + a;
print(b[0]);
print(b[1]);
}
def main() {
    a = [5, 5];
    print(a[0]);
    b = [5, 1] + [3, 2];
    print(b[0]);
    print(b[3]);

    return 0;
}
def main() {
    a = [];
    print(1.1?a); /* print 0 */
    a = a + [1.1];
    print(1.1?a); /* print 1 */
    func(a);
}


def func(a) {
    b = 1.2;
    a = a + [b];
    print(a[1]); /* prints 1.2 */
```

```
    return b;
}
def main() {
    a = [];
    b = a + [5];
    print(b[0]);
}
def main()
{
  print(39 + 3);
  return 0;
}
def main()
{
  print(1 + 2 * 3 + 4);
  return 0;
}
def foo(a)
{
  return a;
}

def main()
{
  a = 42;
  a = a + 5;
  print(a);
  return 0;
}
def main() {
    a= [];
    print(#a);       /* prints 0*/
    a = a + [1];
    print(a[0]);     /* prints 1 */
    print(#a);       /* prints 1 */
    a = a + [2];
    print(#a);       /* prints 2 */
    print(a[0]);     /* prints 1 */
    print(a[1]);     /* prints 2 */
    a = [];
```

```
    print(#a);       /* prints 0 */
    (0<=i<5) { a = a + [i]; }
    print(#a);       /* prints 5 */
    (0 <= i < #a) print(a[i]); /* prints 0 1 2 3 4 */
}
def foo(a){
b = [a];
a = b[0] + a;
return a;
}

def main(){
arr = [1, 2, 3, 4, 5];
res = foo(#arr);
print(res);
}
def main() {
    a = [1,2,3];
    print(#a);
}
def main()
{
    a = [5,3,1,2,4] - [5,3];
    (0<=i<#a) print(a[i]);
}
def main() {
    a = [1,2,3,4];
    a = [1,2,3];
    b = a;
    print(b[1]);
    print(a[1]);
    b[0] = 100;
    print(a[0]);
    func(a);
    print(b[0]);
}


def func(a) {
    a[0] = 4;
```

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

def main()
{
  print(fib(0));
  print(fib(1));
  print(fib(2));
  print(fib(3));
  print(fib(4));
  print(fib(5));
  return 0;
}
def main(){
  (0 < x < 2, 0 < y < 2 | x ==1 && y == 1){
    print(x);
    print(y);
  }
return 0;
}
def main(){
  (0 < x < 5, 2 < y < 12 | x == 3 && y < 5){
    print(x);
    print(y);
}
return 0;
}
def foo(a, b, c){
  (0 <= a <= b, a <= c <= b | true){
  print(a);
}
  return a;
}

def main(){
  foo(1, 3, 2);
```

```
  return 0;
}
def main() {
    a = 6;
    b = [1,2,3,4];
    c = [5,6,7,8];
    print(a?b); /* prints 0 */
    print(a?c); /* prints 1 */
    print(3?b); /* prints 1 */
    print(3?c); /* prints 0 */
    e = 1.1;
    f = [1.1, 2.2];
    print(e?f); /* prints 1 */
}
def main() {
a = [1,2,3];
(0<=i<#a | a[i] == 3) print("hi");
print("bye");
}
def add(a, b)
{
  return a + b;
}

def main()
{
  a = add(39, 3);
  print(a);
  return 0;
}
def printem(a, b, c, d)
{
  print(a);
  print(b);
  print(c);
  print(d);
}

def main()
{
```

```
  printem(42,17,192,8);
  return 0;
}
def add(a, b)
{
  c = a + b;
  return c;
}

def main()
{
  d = add(52, 10);
  print(d);
  return 0;
}
def foo(a)
{
  return a;
}

def main()
{
  return 0;
}
def foo() {}

def bar(a, b, c) { return a + c; }

def main()
{
  print(bar(17, false, 25));
  return 0;
}
a = 1;

def foo(c)
{
  a = c + 42;
}

```

```
def main()
{
  foo(73);
  print(a);
  return 0;
}
def foo(a)
{
  print(a + 3);
}

def main()
{
  foo(40);
  return 0;
}
def main() {
    a = [1,2,3];
    d = func(a); /* passes by reference */
    (0<=i<#d) print(d[i]);

}

def func(a) {
    b = [4,5];
    c = a + b; /* makes a copy */
    return c; /* returns a reference */
}

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

def main()
{
  print(gcd(14,21));
  print(gcd(8,36));
```

```
  print(gcd(99,121));
  return 0;
}
def gcd(a, b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

def main()
{
  print(gcd(2,14));
  print(gcd(3,15));
  print(gcd(99,121));
  return 0;
}
a = 1;
b = 2;

def printa()
{
  print(a);
}

def printk()
{
  print(b);
}

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

def main()
{
  printa();
```

```
  printk();
  incab();
  printa();
  printk();
  return 0;
}
b = true;

def main()
{
  i = 42;
  print(i + i);

  return 0;
}
i = 1;
b = true;
j = 2;

def main()
{
  i = 42;
  j = 10;
  print(i + j);
  return 0;
}
def main()
{
prints("hello world");
}
def main()
{
  if (true) print(42);
  print(17);
  return 0;
}
def main()
{
  if (true) print(42); else print(8);
  print(17);
```

```
  return 0;
}
def main()
{
  if (false) print(42);
  print(17);
  return 0;
}
def cond(b)
{
  if (b)
    x = 42;
  else
    x = 17;
  return x;
}

def main()
{
 print(cond(true));
 print(cond(false));
 return 0;
}
def main()
{
  if (false) print(42); else print(8);
  print(17);
  return 0;
}
def main() {
a = 5;
b = 4;
if (a == b) {
    print(a);
    print(b);
} else
{
print(0);
}
}
```

```
def main() {
    a = [1,2,3, 4, 5];
    b = [4,5,5];
    c = a * b;
    print(#c);
    (0<=i<#c) print(c[i]);
}

def main() {
    a = [1,2,3,4];
    b = [3,4,7,8,9];
    c = a * b;
    print("intersect");
    (0<=i<#c) print(c[i]);
    d = a & b;
    print("union");
    (0<=i<#d) print(d[i]);
}
def main() {

(0<=i<1| true) prints("hello");

print(42);

}

def boo(a) {
    return  a;
}

def main() {
    (1<i<=5, 1<j<= boo(i) | true) {print(i); print(j);}
    a = [1,boo(3),3,boo(3)];
    print("hi");
    print(a[boo(3)]); /* prints 3 */
    print(boo(a[1])); /* prints 3 */
}
def foo(a, b) {
 return a == 1 && b == 1;
}
```

```
741
742 def main() {
743 (0 < x < 2, 0 < y < 2 | foo(x, y)) {
744 print(x); /* prints 1 */
745 print(y); /* prints 1 */
746 } return 0;
747 }
748 def main()
749 {
750     a = [1,2,3];
751     b = [7,3,10];
752     c = [1.1, 2.2, 3.3];
753     d = [4.4, 1.1, 6.6];
754     f = a * b;
755     (0<=i<#f) print(f[i]);
756     f = a + b;
757     (0<=i<#f) print(f[i]);
758     f = a & b;
759     (0<=i<#f) print(f[i]);
760     e = c * d;
761     (0<=i<#e) print(e[i]);
762     e = c + d;
763     (0<=i<#e) print(e[i]);
764     e = c & d;
765     (0<=i<#e) print(e[i]);
766
767 }
768 def main()
769 {
770     a = "hello";
771     b = "hello";
772     c = "hell";
773     print(a == b);      /* prints 1 */
774     print(a == c);      /* prints 0 */
775     d = ["hello", "bye", "hi", "see", "ya"];
776     e = ["hi", "see", "what"];
777     f = d + e;
778     (0<=i<#f) print(f[i]); /* prints hello bye hi see ya hi
        see what */
779     print("hello"?f);      /* prints 1 */
```

```
    g = set(f);
    (0<=i<#g) print(g[i]);  /* prints hello bye hi see ya
   what */
    f = d * e;
    (0<=i<#f) print(f[i]);  /* prints hi see */
    f = d & e;
    (0<=i<#f) print(f[i]);  /* hello bye hi see ya what */
    f = d - e;
    (0<=i<#f) print(f[i]);  /* hello by ya */

}
def foo(b)
{
  i = 42;
  print(i + i);
}

def main()
{
  foo(true);
  return 0;
}
def foo(a, b)
{
  c = a;
  d = b;

  return c + 10;
}

def main() {
 print(foo(37, false));
 return 0;
}
def main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
```

```
  print(99);
  print(1 == 2);
  print(1 == 1);
  print(99);
  print(1 != 2);
  print(1 != 1);
  print(99);
  print(1 < 2);
  print(2 < 1);
  print(99);
  print(1 <= 2);
  print(1 <= 1);
  print(2 <= 1);
  print(99);
  print(1 > 2);
  print(2 > 1);
  print(99);
  print(1 >= 2);
  print(1 >= 1);
  print(2 >= 1);
  return 0;
}
def main()
{
  print(true);
  print(false);
  print(true && true);
  print(true && false);
  print(false && true);
  print(false && false);
  print(true || true);
  print(true || false);
  print(false || true);
  print(false || false);
  print(!false);
  print(!true);
  print(-10);
  print(--42);
}
n = 4;
```

```
859
860 def main()
861 {
862     a = open("input.txt", "r");
863     f = get_input(a);
864     close(a);
865     w = PLA(f);
866     output_results(w);
867 }
868
869 def output_results(w) {
870     a = open("results.txt", "c");
871     write(a, "Optimal Weights: ");
872     (0<=i<#w) write(a, w[i]);
873     close(a);
874 }
875
876 /* Returns the data from file in a set */
877 def get_input(a) {
878     file = read(a);
879     b = split(file, ",");
880     c = ["1"];
881     d = [];
882     (0<=i<#b) {
883         e = split(b[i], " ");
884         new = c + e;
885         d = d + new;
886     }
887     f = [];
888     (0<=i<#d) f = f + [str_to_int(d[i])];
889     return f;
890 }
891
892 /* Performs PLA on the set a */
893 def PLA(a) {
894     w = [0, 0, 0];
895     change = true;
896     while(change) {
897         change = false;
898         (0<=i<#a/n | f(a[n*i:(n*(i+1)-1)], w) * a[n*(i+1)-1
```

```
   ] <= 0) {
899             change = true;
900             adjust(a[n*i:(n*(i+1))], w);
901         }
902     }
903     print("Optimal Weights");
904     (0<=j<#w) print(w[j]);
905     return w;
906 }
907 
908 /* Heuristic function for PLA */
909 def f(data, w) {
910     sum = 0;
911     (0<=i<#data) sum = sum + w[i] * data[i];
912     if (sum > 0) return 1;
913     else {if (sum < 0) return -1; else return 0; }
914 }
915 
916 /* Adjusts the weights */
917 def adjust(data, w) {
918     (0<=i<#w) w[i] = w[i] + data[#data-1] * data[i];
919 }
920 
921 def main() {
922 prints("hello");
923 print(42);
924 }
925 
926 def main(){
927   b = true;
928   print(b);
929   return 0;
930 }
931 def main(){
932         b = ["hi"];
933         print(b[0]);
934 return 0;
935 }
936 def main() {
937     a= [1,2,3,4];
```

```
    (0 <= i < 4| true) print(a[i]);
    print(#a);
    a = [1,2,3];
    print("here");
    print(#a);
    a[1] = a[2];
    print(a[1]);
    b =  a + [9,10];
    (0 <= i < 5| true) print(b[i]);
}
def main() {
    a = [true, true, true];
    print("Len: ");
    print(#a);
    print("false in a");
    print(false?a);
    print("a:");
    (0<=i<#a) print(a[i]);
    a[1] = false;
    print("new a: ");
    (0<=i<#a) print(a[i]);
    print("false in a");
    print(false?a);
    a = set(a);
    print("new a: ");
    (0<=i<#a) print(a[i]);
    print(#a);
}
def main() {
    a = [foo(1), bar(2), cat(3)];
    (0<=i<#a) print(a[i]);
}

def foo(a) {
    return a *2;
}

def bar(b) {
    return b + 2;
}
```

```
978
979 def cat(b) {
980     return b + 3;
981 }
982 def main(){
983         a = [1, 2, 3];
984         print(a[0]);
985         return 0;
986 }
987 def main(){
988    a = [1,2,3,4,5,6,7,8,9,10];
989    b = a;
990    c = b;
991    d = a[1:4];
992    (0<=i<#d) print(d[i]);
993    e = [1.1, 3.3, 4.4, 5.5, 6.6];
994    f = e[0:2];
995    (0<=i<#f) print(f[i]);
996    g = ["hi", "see", "ya", "later"];
997    h = g[1:#g];
998    (0<=i<#h) print(h[i]);
999
1000 }
1001
1002 /*def appendi(arr1, arr2, arr, len1, len2) {
1003     tmp1 = len1;
1004     tmp2 = len2;
1005     (0<=i<len1) arr[i] = arr1[i];
1006     (0<=i<len2) { j = i + len1; arr[j] = arr2[i]; }
1007     return arr;
1008 }*/
1009
1010 def main() {
1011     a = [1,2,3];
1012     b = [4,5];
1013     c = [0, 0, 0, 0, 0];
1014     print(a[1]);
1015     (0<=i<3) print(a[i]);
1016     (0<=i<2) print(b[i]);
1017     (0<=i<5) print(c[i]);
```

```
    print("call");
    appendi(a, b, c, 3, 2);
    print("after");
    (0<=i<5) print(c[i]);
    print("here");
    d = a + b;
    (0<=i<5) print(c[i]);
}
def main()
{
    a = [1,9,6,2,2,3,9];
    d = set_fun(a);
    print("set_fun");
    (0<=i<#d) print(d[i]);
    d = set(a);
    print("set");
    (0<=i<#d) print(d[i]);

}

def set_fun(a) {
    tmp = [];
    (0<=i<#a |  a[i]?tmp == false) tmp = tmp + [a[i]];
    return tmp;
}
def main() {
    a = "hello";
    b = "hello";
    if (a == b) print("hi"); else print("bye");
    return 0;
}
def main(){
        a = "hello";
        b = "hello";
        if ("hello" == "hello") print("hello == hello");
        if (a == "hello") print ("a == hello");
        if (a == b) print ("a == b");
        else print("none");
        return 0;
}
```

```
def main(){
a = 1;
a = -a;
b = false;
b = !false;
print(b);
print(a);
}
def main()
{
  a = 42;
  print(a);
  return 0;
}
a = 1;

def foo(c)
{
  a = c + 42;
}

def main()
{
  foo(73);
  print(a);
  return 0;
}
def main()
{
  i = 5;
  while (i > 0) {
    print(i);
    i = i - 1;
  }
  print(42);
  return 0;
}
def foo(a)
{
  j = 0;
```

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

def main()
{
  print(foo(7));
  return 0;
}
def main(){
  a = 1;
  b = 1;
  while (a != 2){
    while (b != 2)
        {b = b + 1; print(a); print(b);}
  a = a+1;
  }
}
def main() {
i = 5;
i = i-1;
while(i > 0)
{
print(i);
i = i -1 ;
break;
}
print(i);
print(42);
return 0;
}
```