4115 PROGRAMING LANGUAGES AND TRANSLATORS

Monte Carlo Simulation Language

Final Report

DEYZ

Yunling Wang (yw2291)

Chong Zhai (cz2191)

Diego Garcia (dg2275)

Eita Shuto (es2808)

Contents

1 Introduction

1.1 *Overview*

We are studying O'Caml when design this general purpose simulation language. The language aims to simplify the simulation programming with Monte Carlo method, free the programmers to the programming details about the simulation and focus on the model of particular problems. The discussion on generality provided the theoretical base for the feasibility of this idea.

1.2 *Goal*

Monte Carlo Simulation Language-MCSL is a language focusing on simulation problem in many academic areas. The theory of Monte Carlo method has become more and more subtle and is still under development. Our goal is a language which grasps the essential of the language rather than the various detailed implementations. GUI is not considered since this is basically used in the situation where visualization is not kernel.

1.2.1 Sub-algorithms:

We model the work flow of this algorithm as

- generating random numbers,
- evaluate with the sequence of random numbers in the format of vector
- aggregating the simulation results automatically (with the convergences or variational conditions considered simultaneously)

1.2.2 Generation of random numbers

- Uniform distribution:

  Mersenne twister: It is designed with Monte Carlo simulations and other statistical simulations in mind which has long period, high order of dimensional equidistribution and passes numerous stringent tests for statistical randomness.

- Arbitrary distribution:

  Most distribution could be generated by using Uniform [0, 1] random numbers. Algorithms are distribution depended, inverse transformation, acceptance-rejection method, composition method and etc.

We use the GMP-Multiple Precision Arithmetic Library which provides arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. It also has a rich set of efficient functions, its support on *bignum* and random number provide a good opportunity to implement a factorization algorithm shown in the example section.

1.3 Key feature

Most calculations are based on random numbers. Programmer has to do is to specify the algorithm to be used and the type of distribution. The creating of random numbers and way of iterations is taken care of by the language.

1.4 Basic Language Features

1.4.1 Statement

Statement represents a complete instruction. Statements can contain reserved words, operators, and punctuation marks. Examples are shown in Sample Code section.

1.4.2 Data Types

We have defined our data type as follows:
Numeric: Integer, Float, Random Integer, Random Float, Vector
Other: String, Tuple

1.4.3 Reserved Words

The basic vocabulary of MCSL Language consists of a set of pre-defined words, which we call reserved words. Reserved words each have a specific meaning or purpose. Mainly, Basic Reserved Words

int	string	float	vector
randint	randfloat	if	else
do	with	done	

1.4.4 Expression and Operator

a) Mathematical Operators

Arithmetic Operator	Meaning
+	Addition
-	Minus
*	Multiplication
/	Division
%	Reminder
.	Inner Product

b) Relational Operators

Relational Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to

>=	Greater than or equal to
:=	Assign
==	Equal to
!=	Not Equal

c) Logical Operators

Relational Operator Meaning

& Logical AND

| Logical OR

1.4.5 Punctuation Marks

There are a number of punctuation marks to establish statements, define parameters, delimit words, and establish order of precedence.

Symbol Name Description

<< >> starts and ends with vectors.

() Parentheses Group values and forces them to be calculated first

“ ” Quotation Marks Defines a text string

1.4.6 Built in functions

We have a series of Library functions called *Monte-Carlo functions* that are used exclusively for Monte-Carlo simulations.

float Mcaggregate (type1 func, type2 input, type3 iteration-time)

This function performs simulation for *iteration-time* times. It takes one element of *input* each time, apply that to the simulation function *func*, and add the simulation result to the final return value.

int MathFactorial(int num)

This function calculates and returns the factorial of *num*.

int MathPower(int b, int e)

This function calculates and returns the *e*-th power of *b*.

int MathAbs(int num)

This function calculates and returns the absolute value of *num*.

float MathFAbs(float f)

This function calculates and returns the absolute value of *f*.

randFloat RandFloat(float a, float b)

This function defines randFloat type variable which range is from a to b.

randInt RandInt(int a, int b)

This function defines randInt type variable which range is from a to b.

int VectorDimension(vector v)

This function calculates and returns the number of dimensions of the vector *v*.

float VectorLength(vector v)
This function calculates and returns the length of the vector *v*.

1.4.7 User defined functions

In order to support user defined function, we defined the following means to define and declare a function:
type name(type parameter 1, …, type parameter n);

1.5 Sample Code

Sample code to perform operation in MCSL Language:

1.5.1 Generate a random integer/float

```
randFloat  f:= RandFloat (3.0, 2.5)
randInt  i:= RandInt (3, 2)
```

1.5.2 Generate a vector of random integers/floats

```
randInt  i:= RandInt (3, 2)
vector  v:= <<i, i, i >>

randFloat  f:= RandFloat (3.0, 2.5)
vector  v:= <<f, f, f >>
```

2 Language Tutorial

2.1 *Example*

2.1.1 Hello World!

Let's begin with a simple example, "hello world". This is a sample code that displays "hello world!" in a command line.

```
string begin() := "hello world!"
```

Figure 1 Hello World!

They are the basic things to know to implement this tiny code.

- MCSL program should contain a begin function and runs from this function.
- MCSL does not have any explicit return statement, and functions return the value of expression which is contained in this function.
- The returned value of the begin function should be outputted to command line.

2.1.2 Pi Calculation

Next, try Monte Carlo Simulation! This is a small example of Monte Carlo Simulation, π calculation.

```
float inCircle (randFloat x, randFloat y) :=
with
     vector v := <<x, y>>
do
     if VectorLength(v) <= 1
     then 1
     else 0
     endif
done

randFloat domain := RandFloat(0, 1)

float begin(int iterations) :=
     4 * (MCaggregate (inCircle, (domain, domain),
iterations)) / iterations
```

Figure 2 Pi calculation

Three built-in functions are used in this sample code.

- MCaggregate: takes three parameters, evaluation function defined in same source code, parameters that are passed to evaluated function and recursive time. It performs evaluation function for specified times and return accumulation of return values.
- VectorLength: returns the length of the vector.
- RandFloat: defines randomFloat, it takes range of random value.

This is the basic things to know to implement this tiny code.

- The begin function can take arguments from command line.
- Function can contain only one expression in this body.

2.1.3 Useful tips

- A if statement must have else or elseif part. This restriction guarantees that an if statement have type and value. This means a then part and an else part must returns same type. (Auto conversion can be adapted.)
- A logical operator ”and” is single ampersand, not double.
- Semi colon is not needed at end of expression/statement
- Don’t forget termination marks of statements, such as endif and done.

2.2 Compiling and Running

Since MCSL populates Ocaml source code, not executable binary code, users have to do the following step to run their program.

1. Write MCSL source code and save it. For example, helloworld.mcsl

2. Compile a MCSL file to create Ocaml source code, helloworld.ml
$ mclsc helloworld,mcsl

3. Compile a Ocaml source code to create a binary file. If users want to use built-in functions, they have to link our library (libmcsl.cma).
$ ocamlc –o hello helloworld,ml

4. Run!
$ hello

3 Language Manual

3.1 Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

3.1.1 Comments

The token /* introduces a comment, which terminates with the first occurrence of the token */.

3.1.2 Identifiers:

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore _ counts as alphabetic. Upper and lower case letters are considered different.

3.1.3 Keywords

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

int	float	str	vector
randint	randfloat	tuple	list
do	with	done	while
if	else		

3.2 Constants

There are several kinds of constants, as follows:

3.2.1 Integer constants

An integer constant is a sequence of digits.

3.2.3 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision. In this language, some mathematical floating constants are referred by their conventional names in capital case, such as: PI, E. Due to the frequency of their usage, it's supported by the language, not math library.

3.2.3 String constants

A string is a sequence of characters surrounded by double quotes ". A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end.

In a string, the double quotes character " must be preceded by a backslash \ ; in addition, the same escapes as described for character constants may be used.

3.3 DataType

The data types used in MCSL. Data types consist of three forms; a fundamental form, a random form, and a tuple.

3.3.1 Fundamental Form

In MCSL, a fundamental form object is a member of one of the following data types: int, float, vector and string.

int
An int type object represents a 32-bit signed integer value, −2,147,483,648 through 2,147,483,647.

int foo := 27;

float
A float type object represents a double precision floating point number.

float foo := 3.1415;

vector
A vector type object is a combination of one of more float values. A vector type is expressed by sequence of float which is separated by a comma and enclosed by << and >>.

vector foo := << 2.0, 3.2>>;

string
A string type object represent a finite ordered sequence of characters. ASC II set are allowed as a character. A string type is expressed by enclosing with double quotations.

string foo := "hello world!";

3.3.2 Random Form

Random form objects don't have a static value and return different values for each access. These values are generated by pseudo-random algorithm ~~with its distribution defined within the declaration part~~.

randomint
A randomint type object generates different int values for each access.

randomfloat
A randomfloat type object generates different float values for each access.

3.3.3 Tuples

Tuple is a predefined data structure. It is expressed by enclosing with (and), and each node must be delimited by a comma.

3.4 Declaration for variables and functions

Declarations are used to specify the interpretation which MCSL gives to each identifier; the declarations of variables and functions are treated differently.

3.4.1 Variables

All variables should be explicitly declared as below:

type-specifier declarator-assignement-expression

The type-specifier specified the datatype of the variables in the declarator-assignment-expression. The declarator-assignment-expression specifies and a declarator and its value as explained below.

The type specifiers are:

int
float
string
vector
randomint
randomfloat
tuple
list

If the typespecifier is missing from a declaration, it is generally taken to be float.

Declarator-assignment-expression

The declarator-list is a list of declarators with following format:

declarator := value

Declarator

The declarators are names of the variables that are declared.

3.4.2 functions

The declarations of functions have the form

type function-name (parameter-list) := statement

The type is the return type of the function. The function-name is the name of the function. The parameter-list is a list of parameters for the function. They are seperated with comma, and enclosed by "(" and ")". The parameters list has the form

type1 parameter1, type2 parameter2, type3 paratmeter3, ... typeN parameterN

The statement is defined in the section Statement.

4 Project Plan

The MCSL project is composed with project planning, project specification, project development, project debugging and testing.

4.1 Planning:

1. Group Leader: We elected Yunling as our group leader when we first met and it turned out to be our best choice.

2. Brainstorm: We carefully thought about every single previous project showed during the class and also did more investigations on previous projects available on the course website. We spent quite some hours in the first two brainstorm meetings discuss the possibility, the advantage and

disadvantage of various ideas. We focused our attention on 4 proposals: Music composition, Calendar Manipulation, Monte Carlo Simulation and Human Interactive simulation.

3. Making Choice: We met with our TA, and Diego met our professor during the office hour asking about the option on these different projects. After that, we voted for the Monte Carlo simulation language because it's more abstract and generally used. Meanwhile it's easy to implement and demonstrate some algorithms. Another simulation oriented proposal was also good, but we did not have a clear understanding at that moment.

4. Assigning duties: We assigned each member with both common homework and different ones based on different personal preferences and background. It's the best way to make everyone willing to contribute and contribute in a most efficient way. At the meantime, we could learn a lot from each other every meeting.

5. Clarify responsibility, maintain a good schedule: Our group leader made an announcement of responsibility, assignment and corresponding deadline for group member. In the most case, the work load was appropriate for everyone. A reminding email was sent before next meeting to make sure the approach of entire project. It happened that some member was too busy or had difficulty with some assignment. We made adjustment and assigned more people to cooperate with him/her.

6. Timeline: a clear schedule no doubt is crucial for any project. We made sure that everyone has a clear knowledge of important stages for our projects and tried our best to keep the most important stage accomplished on time.

4.2 Project Timeline

Mon 9/8	Team Forming
Fri 9/12	Brain Storming
Tue 9/16	Ask Feedback from TA and Prof
Sat 9/20	Google code SVC created
Mon 9/22	Topic Determined
Wed 9/24	Proposal Submitted
Tue 9/30	Discuss possible application
Fri 10/10	Discuss project's documentation
Thu 10/16	Finish Language Reference Manual
Mon 10/20	Meeting with Professor about details issues
Tue 10/21	LRM Submitted
Thu 10/30	Discuss LRM Feedback, Create Wiki Pages
Thu 11/6	First Parser and Scanner, Final Proposal Added
Tue 11/18	Ast Add into SVN, Modified Parser and Scanner
Sat 11/22	Discuss the details
Wed 11/26	First Working Compiler
Sat 11/29	Discuss the Compiler, Start SAST
Wed 12/10	Implement Complier, SAST, Check File Started

Thu 12/11	PMZ Library Added, Start Final Report
Mon 12/15	First Stable Compiler, More Test Cases
Wed 12/17	Built-in Functions Implemented
Fri 12/19	Subtle Program Created and Tested Demonstrate Project to Professor
	Finish the Final Report

4.3 Roles and Responsibilities

Diego Garcia	Compiler, Interpret, Major System Built-in functions, Source Control setup, Makefile,
Eita Shuto	Ast, Parser, Scanner, Sast, Scoping, Symbol Table, Language Specification,
Yunling Wang	Development Framework Setup ,Test Cases, Random Modulus, Demo Programs
Chong Zhai	Sast, Type Checking, Algorithms, LRM, Final Presentation

4.4 Software Development Environment

All the file are develop with Object Caml, which also provides OCamlyacc.

4.4.1 Operating Systems

Our development was based on Object Caml and its GMP library (which is not easy to compile under windows. Thus we use Linux mostly and Win32 environment sometime. Some modulus could not be compiled under Win32 environment. There is a solution to add GMP with MinGW to use it under windows, but most of the work is done in Linux.

4.4.2 Editor

Three of us use Vi or Vim, one uses another editor, Sakuri, None of use uses IDE.

4.4.3 Subversion (SVN)

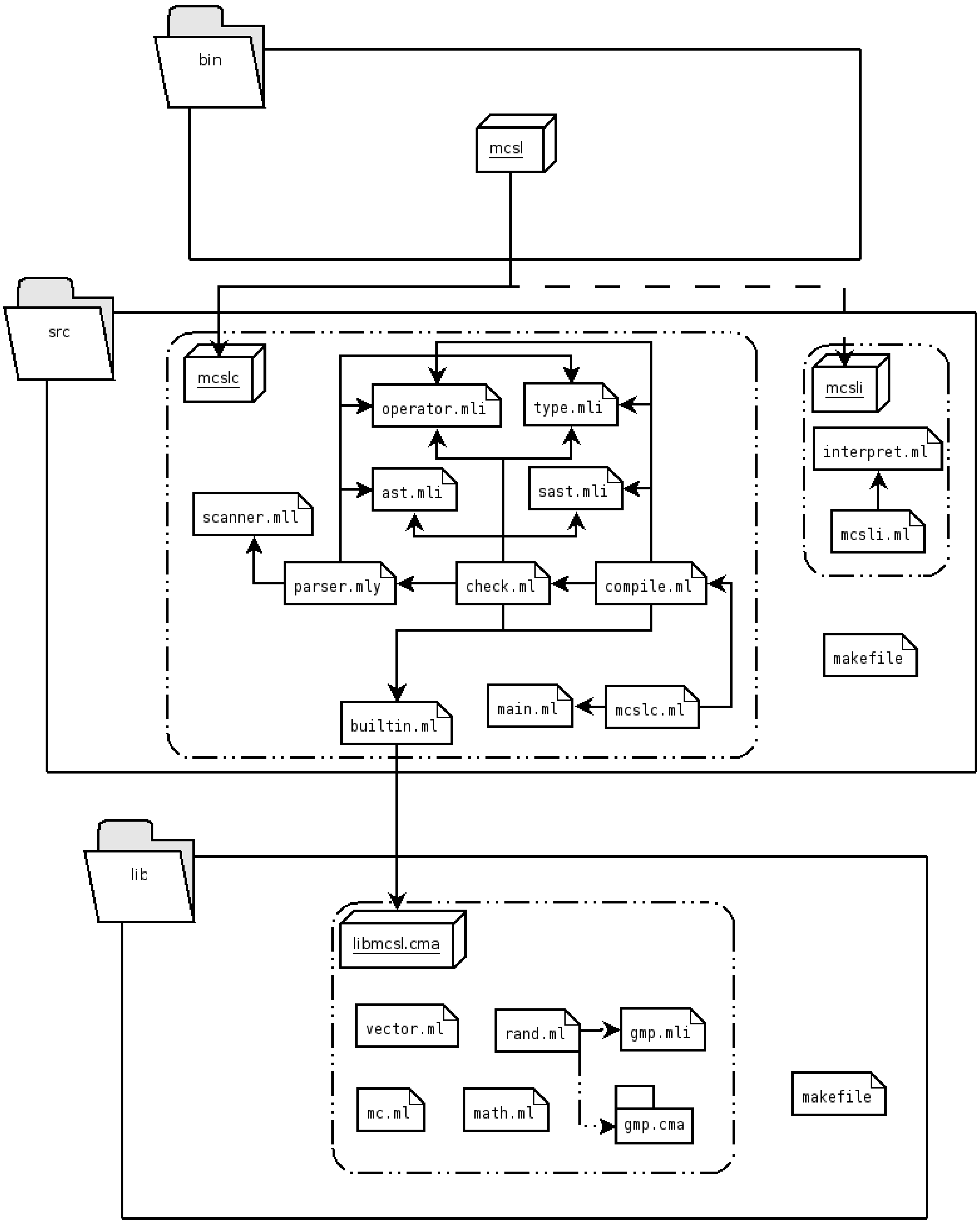
Subversion is an open source application for revision control. We used google code as our repository which is based on SVN.

4.4.4 Bash Shell

We use bash shell to write test cases script.

5 Architectural Design

5.1 Components Diagram



5.2 Compiler Structure

To compile a MCSL source file, it's easiest to use the mcsl executable. However, this is only a front-end script to the real translator. The compiler's main entry point is through mcslc.ml, in the src directory. This file opens the input file, creates an output, and then sequentially calls each layer of the compilation mechanism. First, it calls the lexical scanner, scanner.mll, passing it the source code. From it, a sequence of tokens is returned, which is fed into parser.mly. This parser uses the definitions in operator.mli, type.mli, and ast.mli to create an abstract syntax tree, with nodes for declarations and expressions. The AST is fed then into check.ml through it's chk function. Using the same interfaces than the parser, plus sast.mli, this layer scans the tree checking for proper scope of identifiers and deducing the type of each node. This augmented tree is then passed to compile.ml, which linearly generates corresponding OCaml code to recreate the MCSL functionality. Finally, with all declarations printed out, main.ml is called to complete the translation. It adds code to read in arguments from the command line, convert them to the proper types, and then call begin with them. It then prints out begin's results.

Within MCSL there are a series of core functions available to the user. Whenever check.ml can't find a function in its scope, it checks with builtin.ml to see if it is declared there. If so, the function's information can be retrieved. Similarly, when compile.ml comes across a built-in, it retrieves the name of the real function call from builtin.ml.

Functions declared in builtin.ml are defined in libmcsl.cma, which is in the lib directory. At the moment, libmcsl.cma holds rand.ml, mc.ml, vector.ml and math.ml. It also contains the external library gmp.cma, to which rand.ml is the interface. mc.ml holds the definition of the Monte Carlo functions, designed to easily apply the algorithm to a program. General math functions are in math.ml. In vector.ml are utility functions for vectors. And rand.ml hold the implementations for the random variable types. Also, rand.ml has the only built-in that is implicitly called, when a random type is collapsed into a value.

Once the translation to OCaml is completed, the source is compiled with ocamlc, linking the object with libmcsl.cma, and the MCSL executable is generated.

6 Test Plan

6.1 Test Cases

The test cases generally have 3 categories: basic tests (including all arithmetic operators, statements and expression evaluations), random tests (functions that call the random module), and advanced tests (programs that do actual simulation under a well-defined context).

Here are the two typical test cases: nest.mcsl and fac.mcsl.

6.1.1 nest.mcsl

Mostly tested the scoping rules, including the opening scoping, static scoping, nested scoping, as well as our special scoping expression "with…do…done".
This program actually succeeded in breaking the compiler the first time it ran.

```
                    ##===================nest.mcsl=================##
int begin() :=
with
        int x := 3
do
        if x < 4
        then
                if
                        with
                                int y := 12
                        do
                                y/x
                        done
                then  1
                else  -1
                endif
        else
                0
        endif
done

                    ##===================nest.ml=================##
let rec begin' = ( let x' = 3 in
 (if ( if (float_of_int x' ) < (float_of_int 4 ) then 1 else 0 ) = 0 then (
 0
) else (
 (if ( let y' = 12 in
 ( y' / x' ) ) = 0 then (
 ( - 1 )
) else (
 1
))
)) ) ;;

let _ret = begin'
```

```
 in print_endline
(string_of_int _ret)
```

6.1.2 fac.mcsl

This program is a real simulation program under a well-defined context. It can factorialize an nonnegative integer. This shows how our language can be used in actual simulation applications.

```
                                    ##===================fac.mcsl=================##
int isprime(int n):=
if n!=2 & n%2 == 0
        then 0
else
        with
                int checkprime(int n,int i):=
                        if i*i > n then 1
                        elseif n%i == 0 then 0
                        else  checkprime(n, i+2)
                        endif
        do
                checkprime(n, 3)
        done
endif

int gcd(int a, int b):=
if a == b
        then a
elseif a > b
        then gcd(a-b, b)
else
        gcd(b-a, a)
endif

int makeodd(int n):=
if n%2==0
        then makeodd(n/2)
else
        n
endif

string factorial(int n, int b, int k):=
with
        string str := ""
        int tmp := n-1
        randInt iran := RandInt(0,tmp-1)
        int a :=
                if iran <= 1
                then 2
                else iran
                endif
        int power := MathPower(a, k)
        int res := gcd(MathAbs((power)%n-1), n)
        int change := res > 1 & isprime(res)
        int n :=
```

```
                    if change
                    then n/res
                    else n
                    endif
          string str := if change then
                              str + " " +res
                    else
                              str
                    endif
do
          if isprime(n)
                    then n+" "+str      /*print N */
          elseif n==1
                    then str
          else
                    str+" "+factorial(n, b, k)
          endif
done

string fact(int n, int b):=
with
          int k := MathFactorial(b)
do
          factorial(n, b, k)
done


string begin(int n):=
with
          int b:= 6
          int n := makeodd(n)
do
          if n==1
          then ""
          elseif  isprime(n)
          then n
          else
        fact(n,b)
          endif
done
```

##===================fac.ml=================##

```
let rec isprime' (n' ) = (if ( if (float_of_int ( if (float_of_int n' ) <> (float_of_int ( if ( 2 <> 0) && ( ( n' mod 2 ) <> 0) then 1
else 0 ) ) then 1 else 0 ) ) = (float_of_int 0 ) then 1 else 0 ) = 0 then (
 ( let rec checkprime' (n' , i' ) = (if ( if (float_of_int ( i' * i' ) ) > (float_of_int n' ) then 1 else 0 ) = 0 then (
 (if ( if (float_of_int ( n' mod i' ) ) = (float_of_int 0 ) then 1 else 0 ) = 0 then (
 (checkprime' ( n' , ( i' + 2 ) ) )
) else (
 0
))
) else (
 1
)) in
 (checkprime' ( n' , 3 ) ) )
) else (
 0
)) ;;
```

```
let rec gcd' (a' , b' ) = (if ( if (float_of_int a' ) = (float_of_int b' ) then 1 else 0 ) = 0 then (
(if ( if (float_of_int a' ) > (float_of_int b' ) then 1 else 0 ) = 0 then (
(gcd' ( ( b' - a' ) , a' ) )
) else (
(gcd' ( ( a' - b' ) , b' ) )
))
) else (
a'
)) ;;
let rec makeodd' (n' ) = (if ( if (float_of_int ( n' mod 2 ) ) = (float_of_int 0 ) then 1 else 0 ) = 0 then (
n'
) else (
(makeodd' ( ( n' / 2 ) ) )
)) ;;
let rec factorial' (n' , b' , k' ) = ( let str' = "" in
let tmp' = ( n' - 1 ) in
let iran' = (Rand.intRng ( 0 , ( tmp' - 1 ) ) ) in
let a' = (if ( if (float_of_int  (Rand.getRandInt  iran' ) ) <= (float_of_int 1 ) then 1 else 0 ) = 0 then (
(Rand.getRandInt  iran' )
) else (
2
)) in
let power' = (Math.pow ( a' , k' ) ) in
let res' = (gcd' ( (Math.abs ( ( ( power' mod n' ) - 1 ) ) ) , n' ) ) in
let change' = ( if ( ( if (float_of_int res' ) > (float_of_int 1 ) then 1 else 0 ) <> 0) && ( (isprime' ( res' ) ) <> 0) then 1 else 0 )
in
let n' = (if change' = 0 then (
n'
) else (
( n' / res' )
)) in
let str' = (if change' = 0 then (
str'
) else (
( ( str' ^ " " ) ^ (string_of_int res' ) )
)) in
(if (isprime' ( n' ) ) = 0 then (
(if ( if (float_of_int n' ) = (float_of_int 1 ) then 1 else 0 ) = 0 then (
( ( str' ^ " " ) ^ (factorial' ( n' , b' , k' ) ) )
) else (
str'
))
) else (
( ( (string_of_int n' ) ^ " " ) ^ str' )
)) ) ;;
let rec fact' (n' , b' ) = ( let k' = (Math.factorial ( b' ) ) in
(factorial' ( n' , b' , k' ) ) ) ;;
let rec begin' (n' ) = ( let b' = 6 in
let n' = (makeodd' ( n' ) ) in
(if ( if (float_of_int n' ) = (float_of_int 1 ) then 1 else 0 ) = 0 then (
(if (isprime' ( n' ) ) = 0 then (
(fact' ( n' , b' ) )
) else (
(string_of_int n' )
))
) else (
```

```
 ""
)) ) ;;

let _param1 = int_of_string Sys.argv.(1);;
let _ret = begin'
(
_param1)
 in print_endline
_ret
```

6.2 Using of script in Testing

Scripts are used in multiple places during testing. First, There is a script that automatically compiles the {sourcecode}.mcsl into {sourcecode}.ml then to {sourcecode}, and executes the executable at last; Second, there is a test script that calls the first script to compile and run the program before comparing their outputs to the expected ones.

6.2.1 mcsl.sh

This is the first script that compile and execute the source file automatically. Parts of the codes are shown below:

```
##==================mcsl.sh=================##
#!/bin/bash

# MCSL compiling script.  Automates steps for converting a mcsl source file
# into an executable, or an in between state.

Usage () { cat; } <<doc
Usage: $CMD [options] filename
mcsl is a frontend to mcslc and mcsli, respectively the Monte Carlo Simulation
Language's compiler and interpreter.  By default, it will compile the input
file into an executable.  Use the options to change its behaviour.
Options:
   -C <file>       Use <file> as mcsl compiler
   -h              Print usage and exit
   -o <file>       Place output into <file>
   -t              Only translate to ocaml, don't compile
doc

Error () {
   rm -f $RMLIST &>/dev/null
   echo "$CMD: ${1:-"error"}" >&2
   exit ${2:-1}
}

# Get compiler command and directory
CMD=${0##*/}
DIR=${0%/*}

# Defaults
MCSLC="$DIR/../src/mcslc"
```

```
MCSLI="$DIR/../src/mcsli"
MCLIB="$DIR/../lib"
COMPILE=true
LIBS="libmcsl.cma"
RMLIST=""

# Minimal check
if [[ -z $1 ]] ;
then Error "no input files";
fi

# Scan arguments
while [[ -n $1 ]] ;
do
  case $1 in
  # Use an alternative compiler executable
        -C)
        if [[ -z $2 ]] ;
        then Error "no file for -C option"
        fi
        MCSLC=$2
        shift 2
        ;;

  # Print usage and exit
        -h)
        Usage
        exit 0
        ;;

  # Set an output file
        -o)
        if [[ -z $2 ]] ;
        then Error "no file for -o option"
        fi
        OUT=$2
        shift 2
        ;;

  # No compilation
        -t)
        COMPILE=
        shift
        ;;

  # Get input file and check for existance and extension
        *)
        if [[ -n $SRC ]] ;
        then Error "too many input files";
        fi

        SRC=$1
        if [[ ! -r $SRC ]] ;
        then Error "can't read file: $SRC";
        fi
```

```
            BASE=${1%.*}
            BASE=${BASE##*/}
            EXT=${1##*.}
            case $EXT in
              "mcsl")
              ;;
              *)
              Error "unknown filetype: $SRC"
              ;;
            esac
            shift
            ;;
    esac;
done

# Translate mcsl to ml
if [[ ! -x $MCSLC ]] ;
then Error "can't execute compiler"
fi
RMLIST="$RMLIST $BASE.ml"
$MCSLC $SRC
if [[ $? -ne 0 ]] ;
then
    Error
fi
if [[ ! $COMPILE ]] ;
then
    if [[ -n $OUT ]] ;
    then mv $BASE.ml $OUT
    fi
    exit 0
fi

# Compile ml to executable
OCAMLC=$(which ocamlc)
if [[ $? -ne 0 ]] ;
then Error "can't find ocamlc"
fi
$OCAMLC -o ${OUT:=$BASE} -I $MCLIB $LIBS $BASE.ml
if [[ $? -ne 0 ]] ;
then Error
fi
RMLIST="$RMLIST $BASE.cmo $BASE.cmi"
rm -f $RMLIST &> /dev/null
./$OUT
```

6.2.2 test.sh

All the tests cases are called and evaluated by a bash script, which automatically compares the program outputs and the expected results. The script is modified from the microc test programs by professor Edwards on the course websites .
Part of the script is as below:

##==================test.sh=================##

```
#!/bin/sh

MCSL="./mcsl"
Compare() {
   generatedfiles="$generatedfiles $3"
   echo diff -b $1 $2 ">" $3 1>&2
   diff -b "$1" "$2" > "$3" 2>&1 || {
          SignalError "$1 differs"
          echo "FAILED $1 differs from $2" 1>&2
   }
}

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

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

   echo -n "$basename..."

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

   generatedfiles="${basename}.out" &&
   Run "$MCSL" $1 ">" ${basename}.out &&
   Compare ${basename}.out ${reffile}.out ${basename}.out.diff

   # Report the status and clean up the generated files

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

files="basictests/*.mcsl"
```

```
for file in $files
do
	Check $file 2>> $globallog
done
```

7. Lessons Learned

Eita Shuto

First of all, this project gave me a good chance to consider programming languages. Until this project (and this course), I had regarded the programming language as only rules, and was interested in only "what/how", not "why". When we tried changing some part of current language and create new one, it sometimes caused unexpected side effect in other place, even if it looked reasonable at first. Through such trial and error, I understood why current languages have such syntax or semantics.

In addition, this was my first project which used a functional language. I know that there is a bunch of bugs in the world and many software engineers are suffering from them. If functional language is a good solution to solve this problem, it is really great, even if it is still difficult for me and writing code with functional languages takes more than three times than writing code with other languages that are familiar to me. Now, I begin to be interested in other functional programming languages, such as Heskell and F#.

Finally, if I can give one advice to teams in the future, I will say that you should decrease the number of data type. Supporting many data types is really tiring and requires many tests. If I would define new programming language again, this language should have only one data type!

Chong Zhai

Being a first semester student without computer program background taking five courses, I think nothing is more important than time management. A good schedule with moderate work helps a lot. In fact, our group leader assigned each member based on the background which toke everyone's best potential. This is the basic principle when distributes human resources as a group leader but it is not easy. Yunling did a good job with it. Next thing I learned is cooperation. I could say this project is still doable individually. What distinguishes a group project with CVN student is neither the size of the project, nor the quality of the project, but learning to work with other people who may present a large diversity of productivity, responsibility and personality. This is one of the most important things in both academic and nonacademic career. Another thing I learned is to be realistic. This is also the only thing my first English teacher tells me. People tend to have big expectation on himself/herself. In this case, quite a lot of beautiful and useful ideas we put on our waiting lists are not implemented given the time. Even the top implementation was cut when deadline approaches. At last, I think I learned to implement the basic algorithm in computer science – divide and conquer in practice. It's more often saving time to break a big project into minuteness components and conquer them one by one. In this project, during the project it happened that we thought something was trivial but it turned out not unless we understand each small component. Technically, what I learned is a functional language with a functional programming style plus the knowledge of compiler. Talking about this project, I also learned to use the abstract theory into real implementation, more importantly, being critical whenever comes across a new language. In all, I suggest future project groups start early, decide quickly, dream less and make small progress first.

Diego Garcia

Sometimes, all the planning session you have simply aren't a substitute for experience. We took very long to get started with the code, mainly because we wanted to have all our bases covered. However, once doing the implementation, all manner of problems completely beyond our expectations plagued us. In hindsight, there are many things I would have done differently.

The basic concept of our language is sound, even when the implementation wasn't. Given another week or so, I'd be tempted to rewrite the whole of the compiler from scratch, and probably a new series of nuances would then surface.

Yunling Wang

This is really an unforgettable experience of implementing our Monte Carlo Simulation Language. There are some interesting points that I got during this process.

First, I got to really understand the functional style of programming. We wanted to implement a hybrid style language that has both elements from C and Ocaml, with the mainstream style as functional style. Interestingly, as time goes on, we gradually got rid of the C elements and moved bit by bit to adopt the Ocaml like functional language elements. At last, we came to the conclusion that it would never be harmony to have both C and Ocaml elements in a single language, so we simply abandoned lots of C staffs, like looping statement, variable re-evaluation. Meanwhile, we start to reconstruct our language to make it more functional-like: changing all the branching and scoping statements as well as the function calls to expressions, and deleting expression statement completely, leaving only the function and variable declaration as the only types of statements. This mostly resolved our problems of failing to evaluate the types of the statements when doing automatic type conversion.

Second, I implemented the GMP random number interface module for our language. This is a well-written arbitrary precision arithmetic operation library that is widely used in simulation process. At first, I was trying to figure out ways of linking the C library directly. Later, however, it turns out the Ocaml-Gmp interface is already implemented by David Monniaux in a package called mlgmp. So the task becomes writing a random module that calls the mlgmp interface to implement the random-number-generation functionality. The basic design idea of the random module is to create a random number generator that keeps all the restrictions for the number generation on its declaration, and returns an actual random value every time this number is evaluated as an integer or float.

Third, I did all of the test cases for our language. Most of the testing error occurred in type conversions and checkings. Basically, there are several kinds of conversions: those between int, string, float, vector and tuple; those between random number and the rest of the tests; those occur in function argument evaluation or return type evaluation; Second, the scoping rules are also import to test: static/dynamic, opening/close, and those nested in branching and special scoping expressions (most of our error here involves with the mismatch of parenthesis in generated .ml code).

At last, as the team leader, I learnt a lot about how to organize the whole developing process of the team. I tried my best in assigning tasks to cater for the specialties and skills of the team members, and in setting periodic goals to ensure progress of the development of the project.

8. Appendix

(**)
(* src/: Main compiler implementation *)
(**)

```
(******************************************************************************)
(* src/scanner.mll: Token scanner                                             *)
(******************************************************************************)
{ open Parser }
let digit = ['0'-'9']

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"     { comment lexbuf }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LLIST }
| ']'      { RLIST }
| "<<"     { LVECTOR }
| ">>"     { RVECTOR }
| ';'      { SEMI }
| ','      { COMMA }
| '!'      { NOT }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { REMIND }
| '&'      { LAND }
| '|'      { LOR }
| '.'      { INNERP }
| '"'      { QUOT }
| ":="     { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "elseif" { ELSEIF }
| "endif"  { ENDIF }
| "with"   { WITH }
| "do"     { DO }
| "done"   { DONE }
| "then"   { THEN }
| "float"  { FLOAT }
| "int"    { INT }
| "vector" { VECTOR }
| "string" { STR }
| "randInt "   { RINT }
```

```
| "randFloat"   { RFLOAT }
| "list"        { LIST }
| digit+ as lxm { LINT(int_of_string lxm) }
| digit+ ('.' digit*)? (['e' 'E'] ['+' '-']? digit+)? as lxm { LFLOAT(float_of_string (lxm)) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"' ([^'"']* as lxm) '"' { LSTR(lxm) }
| eof      { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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


(******************************************************************************)
(* src/parser.mly: Parser. Returns AST tree.                                  *)
(******************************************************************************)
%{ open Ast %}
%{ open Operator %}
%{ open Type %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LLIST RLIST LVECTOR RVECTOR COMMA
%token MINUS NOT
%token PLUS MINUS TIMES DIVIDE ASSIGN INNERP REMIND
%token EQ NEQ LT LEQ GT GEQ LAND LOR
%token IF THEN ELSE ELSEIF ENDIF WITH DO DONE
%token INT FLOAT VECTOR LIST RINT RFLOAT STR QUOT
%token <int> LINT
%token <float> LFLOAT
%token <string> ID
%token <string> LSTR
%token EOF

%left NOT
%left EQ NEQ
%left LAND LOR
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE INNERP REMIND

%start program
%type <Ast.program> program

%%

program:
  /* nothing */ { [] }
 | dclr program { ($1 :: $2) }

formals_opt:
   LPAREN RPAREN { [] }
  | LPAREN formal_list RPAREN  { List.rev $2 }

formal_list:
   formal  { [$1] }
  | formal_list COMMA formal { $3 :: $1 }

formal:
```

```
   tp ID { ($1, $2) }

stmt_list:
   stmt  { [$1] }
  | stmt_list stmt { $1@[$2] }

dclr:
   tp ID formals_opt ASSIGN expr { FDclr($1, $2, $3, $5) }
  | tp ID ASSIGN expr { VDclr($1, $2, $4) }


stmt:
   dclr { $1 }

else_list:
   ELSE expr { $2 }
  | ELSEIF expr THEN expr else_list { If($2, $4, $5) }

expr:
   LINT           { LInt($1) }
  | LFLOAT         { LFloat($1) }
  | LBRACE expr_list RBRACE    { LList($2) }
  | LVECTOR expr_list RVECTOR   { LVctr($2) }
  | LPAREN expr_tuple RPAREN    { LTple($2) }
  | LSTR           { LString($1) }
  | ID             { Id($1) }
  | MINUS expr      { Uop(Minus, $2) }
  | NOT expr        { Uop(Not, $2) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr REMIND expr { Binop($1, Rmndr, $3) }
  | expr INNERP expr { Binop($1, Innrp, $3) }
  | expr LAND   expr { Binop($1, Land,  $3) }
  | expr LOR    expr { Binop($1, Lor,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | ID LLIST expr RLIST { Elmt($1, $3) }
  | IF expr THEN expr else_list ENDIF { If($2, $4, $5) }
  | WITH stmt_list DO expr DONE { Scope($2, $4) }

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

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

actuals_opt:
   /* nothing */ { [] }
```

```
  | actuals_list  { List.rev $1 }

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

tp:
    FLOAT  { Float }
  | INT    { Int }
  | VECTOR { Vector(0) }
  | RINT   { RInt }
  | RFLOAT { RFloat }
  | LIST   { List }
  | STR    { Str }
```

```
(*****************************************************************************)
(* src/ast.mli: AST tree definition.                                         *)
(*****************************************************************************)
open Type
open Operator

type expr =
    | LInt of int
    | LFloat of float
    | LList of expr list
    | LVctr of expr list
    | LTple of expr list
    | LString of string
    | Uop of uop * expr
    | Id of string
    | Binop of expr * binop * expr
    | Call of string * expr list
    | Elmt of string * expr
    | If of expr * expr * expr
    | Scope of stmt list * expr
   (*| Noexpr*)

and stmt =
    | FDclr of tp * string * (tp * string) list * expr
    | VDclr of tp * string * expr
    | While of stmt list * expr

type program = stmt list
```

```
(*****************************************************************************)
(* src/operator.mli: Operator types definition.                              *)
(*****************************************************************************)
type binop = Add | Sub | Mult | Div | Rmndr | Innrp | Equal | Neq | Less | Leq |
Greater | Geq | Land | Lor

type uop = Minus | Not
```

```
(*****************************************************************************)
(* src/types.mli: Variable types definition.                                 *)
(*****************************************************************************)
```

```
type tp =
   | Float
   | Int
   | Vector of int
   | RInt
   | RFloat
   | List
   | Str
   | Tuple
   | Func
```

```
(*********************************************************************************)
(* src/check.ml:  AST to SAST converter.  Checks types and scope.             *)
(*********************************************************************************)
open Sast
open Ast
open Type
open Operator

module SymTbl = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)


(* (from type, to type)  *)
let rec checkType types = match types with
   | (_, Int) ->
      (match types with
       | (Int, _)|(Float, _)|(RInt, _)|(RFloat,_) -> Int
       | (_, _)   -> failwith "Type Convert Mismatch (Int)"
       )
   | (_, Float) ->
      (match types with
       | (Int, _)|(Float, _)|(RInt, _)|(RFloat,_) -> Float
       | (_, _)   -> failwith "Type Convert Mismatch (Float)"
       )
   | (_, RInt) ->
      (match types with
       | (RInt, _) -> RInt
       | (_, _)   -> failwith "Type Convert Mismatch (RInt)"
       )
   | (_ , RFloat) ->
      (match types with
       | (RFloat, _) -> RFloat
       | (_, _)   -> failwith "Type Convert Mismatch (RFloat)"
       )
   | (_ , Tuple) -> Tuple
   | (_, Str) ->
      (match types with
       | (Str, _)|(Int, _)|(Float, _)|(RInt, _)|(RFloat,_)|(Vector 0,_) -> Str
       | (_, _)   -> failwith "Type Convert Mismatch (Str)"
       )
   | (Vector 0, Vector 0) -> Vector 0
   | (_, Func) -> Func
   | (_, _)   -> failwith "Type Convert Mismatch"
```

```
;;

let rec canConversion types = match types with
   | (Int, _)|(RInt, _) ->
      (match types with
       | (_, Int)|(_, RInt) -> Int
       | (_, Float)|(_, RFloat) -> Float
       | (_, Str) -> Str
       | (_, _)   -> failwith "If/Else Mismatch (Int/RInt)"
       )
   | (Float, _)|(RFloat, _) ->
      (match types with
       | (_, Int)|(_, RInt) -> Float
       | (_, Float)|(_, RFloat) -> Float
       | (_, Str) -> Str
       | (_, _)   -> failwith "If/Else Mismatch (Float)"
       )
   | (Str, _) ->
      (match types with
       | (_, Str)|(_, Int)|(_, RInt)|(_, Float)|(_, RFloat)|(_, Vector 0) -> Str
       | (_, _)   -> failwith "If/Else Mismatch (Str)"
       )
   | (Vector 0, Vector 0) -> Vector 0
   | (Vector 0, Str) -> Str
   | (Tuple, Tuple) -> Tuple
   | (_, _) -> failwith "Cannot convert"
;;

let convert_arg_type to_args from_args =
   let (v, typ) = from_args in (from_args, (checkType (typ, to_args)))
;;

(* find function and return a return type *)
let rec findfun id = function
   | [] ->
      if Builtin.exists id
         then Builtin.get_ret_type id
         else failwith ("Undeclared Function:" ^ id)
   | loc::scp ->
      let (vars,funs) = loc in
      if SymTbl.mem id funs
         then
            let (typ, arg_typ) = SymTbl.find id funs
            in typ
         else findfun id scp
;;

(* find a function from name and return argument types *)
let rec findfunargs id = function
   | [] ->
      if Builtin.exists id
         then Builtin.get_arg_types id
         else failwith ("Undeclared Function:" ^ id)
   | loc::scp ->
      let (vars,funs) = loc in
      if SymTbl.mem id funs
         then
            let (typ, arg_typ) = SymTbl.find id funs
```

```
            in arg_typ
         else findfunargs id scp
;;


(* find a function and return variable types *)
let rec findvar id = function
   | [] -> failwith ("Undeclared Variable:" ^ id)
   | loc::scp ->
      let (vars,funs) = loc in
      if SymTbl.mem id vars
         then SymTbl.find id vars
         else if SymTbl.mem id funs
            then let (typ, args) = SymTbl.find id funs
               in Func (* typ *)          (* TODO:a type of function variables may be function... *)
            else findvar id scp
;;

let chk ast =
   let rec expr env = function
           Ast.LInt e -> Sast.LInt e, Int
         | Ast.LFloat e -> Sast.LFloat e, Float
         | Ast.Binop(e1, op, e2) ->
            let e1 = expr env e1
            and e2 = expr env e2 in
            let _, t1 = e1
            and _, t2 = e2
            in (match op with
            |  Add -> (match (t1,t2) with
               | (Int, _) | (RInt, _) -> (match (t1,t2) with
                  | (_, Int) | (_, RInt)     -> Sast.Binop(e1, op, e2), Int
                  | (_, Float) | (_, RFloat)  -> Sast.Binop(e1, op, e2), Float
                  | (_, Str)  -> Sast.Binop(e1, op, e2), Str
                  | (_, _)              -> failwith "Unsupported Add")
               | (Float, _) | (RFloat, _) -> (match (t1,t2) with
                  | (_, Int) | (_, RInt)|(_, Float) | (_, RFloat) -> Sast.Binop(e1, op, e2), Float
                  | (_, Str)  -> Sast.Binop(e1, op, e2), Str
                  | (_, _)              -> failwith "Unsupported Add")
               | (Vector a, _) -> (match (t1,t2) with
                  | (_, Vector b) -> Sast.Binop(e1, op, e2), Vector a
                  | (_, Str)  -> Sast.Binop(e1, op, e2), Str
                  | (_, _)              -> failwith "Unsupported Add")
               | (Str, _) -> (match (t1,t2) with
                  | (_, Int) | (_, RInt)|(_, Float) | (_, RFloat)|(_, Str)|(_, Vector 0) -> Sast.Binop(e1, op, e2), Str
                  | (_, _)              -> failwith "Unsupported Add")
               | (_, _) -> failwith "Unsupported Add/Sub")

            |  Sub -> (match (t1,t2) with
               | (Int, _) | (RInt, _) -> (match (t1,t2) with
                  | (_, Int) | (_, RInt)     -> Sast.Binop(e1, op, e2), Int
                  | (_, Float) | (_, RFloat)  -> Sast.Binop(e1, op, e2), Float
                  | (_, _)              -> failwith "Unsupported Sub")
               | (Float, _) | (RFloat, _) -> (match (t1,t2) with
                  | (_, Int) | (_, RInt)|(_, Float) | (_, RFloat) -> Sast.Binop(e1, op, e2), Float
                  | (_, _)              -> failwith "Unsupported Sub")
               | (Vector a, _) -> (match (t1,t2) with
                  | (_, Vector b) -> Sast.Binop(e1, op, e2), Vector a
                  | (_, _)              -> failwith "Unsupported Sub")
```

```
      | (_, _) -> failwith "Unsupported Add/Sub")

    | Equal|Neq|Greater|Geq|Leq|Less|Land|Lor ->
    (* use Int instead of boolean *)
    (match (t1,t2) with
        | (Int, _) | (RInt, _)| (Float, _) | (RFloat, _) -> (match (t1,t2) with
           | (_, Int) | (_, RInt)| (_, Float) | (_, RFloat) -> Sast.Binop(e1, op, e2), Int
           | (_, _) -> failwith "Unsupported Compare")
        | (_, _) -> failwith "Unsupported Compare")

    | Mult -> (match (t1, t2) with
        | (Int, _) | (RInt, _) -> (match (t1,t2) with
           | (_, Int) | (_, RInt)      -> Sast.Binop(e1, op, e2), Int
           | (_, Float) | (_, RFloat)  -> Sast.Binop(e1, op, e2), Float
           | (_, Vector a)             -> Sast.Binop(e1, op, e2), Vector a
           | (_, _)                    -> failwith "Unsupported Mult")
        | (Float, _) | (RFloat, _) -> (match (t1,t2) with
           | (_, Int) | (_, RInt)|(_, Float) | (_, RFloat) -> Sast.Binop(e1, op, e2), Float
           | (_, _)                    -> failwith "Unsupported Mult")
           | (_, Vector a)             -> Sast.Binop(e1, op, e2), Vector a
        | (_, _) -> failwith "Unsupported Mult")

    | Div -> (match (t1, t2) with
        | (Int, _) | (RInt, _) -> (match (t1,t2) with
           | (_, Int) | (_, RInt)      -> Sast.Binop(e1, op, e2), Int
           | (_, Float) | (_, RFloat)  -> Sast.Binop(e1, op, e2), Float
           | (_, _)                    -> failwith "Unsupported Div")
        | (Float, _) | (RFloat, _) -> (match (t1,t2) with
           | (_, Int) | (_, RInt)      -> Sast.Binop(e1, op, e2), Float
           | (_, Float) | (_, RFloat)  -> Sast.Binop(e1, op, e2), Float
           | (_, _)                    -> failwith "Unsupported Div")
        | (_, _) -> failwith "Unsupported Div")

    | Rmndr -> (match (t1, t2) with
        | (Int, _) | (RInt, _) -> (match (t1,t2) with
           | (_, Int) | (_, RInt)      -> Sast.Binop(e1, op, e2), Int
           | (_, _)                    -> failwith "Unsupported Reminder")
        | (_, _) -> failwith "Unsupported Reminder")

    | Innrp -> (match (t1,t2) with
        | (Vector a, Vector b)  -> Sast.Binop(e1, op, e2), Float
        | (_, _) -> failwith "Unsupported Innrp")
    )

 | Ast.Uop(op, e) ->
            let e = expr env e in
            let _, t = e in
            (match op with
            | Minus -> (match t with
            | Int|Float|RInt|RFloat|(Vector _) -> Sast.Uop(op, e), t
            | _ -> failwith "Unsupported Minus")

            (* Right now, Not could be operated on any type
             * Everything could be regarded as boolean 1 expect 0*)
            | Not -> Sast.Uop(op, e), t)
| Ast.LString str -> Sast.LString str, Str
| Ast.LList e ->
           Sast.LList (List.fold_left (fun ls ex -> (expr env ex)::ls)
```

```
            [] e), List
    | Ast.LVctr e -> Sast.LVctr (List.fold_left (fun ls ex -> (expr env ex)::ls)
            [] e), Vector (List.length e)
    | Ast.LTple e -> Sast.LTple (List.fold_left (fun ls ex -> (expr env ex)::ls)
            [] e), Tuple
    | Ast.Elmt (s, e) ->
            let e = expr env e in Sast.Elmt (s, e), Int
    | Ast.Id e -> Sast.Id e, (findvar e env)

    | Ast.Call (func_name, args) ->
       let sast_args = List.fold_left(fun ls ex -> ls@[(expr env ex)]) [] args in         (* Types and value in call *)
       let fun_args =  (findfunargs func_name env) in                                   (* Types in decrlation *)
       let cnvt_new_args = (List.map2 convert_arg_type fun_args sast_args) in
       Sast.Call(func_name, cnvt_new_args), findfun func_name env

    | Ast.If(e, e1, e2) ->
          let e = expr env e in
          let _, t = e in
          (match t with
             | Int ->
               let (eif, tpif) = (expr env e1) in
               let (eelse, tpelse) = (expr env e2) in
               let typ = canConversion(tpif, tpelse) in
                  Sast.If(e, (eif, tpif), (eelse, tpelse)), typ
             | _ -> failwith "Predicate of if must be integer"
          )

    | Ast.Scope (inits, body) ->
          let env = (SymTbl.empty, SymTbl.empty)::env
          in
          let (s1' , env)=
             List.fold_left
               (fun (st, env') s ->
                  let (stmt, env) = (stmt env' s)
                  in
                  (st@[stmt], env)
               )
               ([], env) inits
          in
          let (body, typ) = (expr env body)
          in
          Sast.Scope (s1', (body, typ)), typ

and
stmt env = function

    | Ast.VDclr(t, str, e) -> (match env with
       [] -> failwith "empty scope in FDclr"
       | (lvars, lfuns)::globals ->
          let lvars = SymTbl.add str t lvars in
          let env = (lvars, lfuns)::globals in (
       Sast.VDclr(t, str, expr env e), env))
    | Ast.While (s, e) -> failwith "While is not supported!"
    | Ast.FDclr (t1, str, args, e) -> (match env with
       [] -> failwith "empty scope in FDclr"
       | (lvars, lfuns)::globals ->
          let args_types = List.fold_left (fun acc (typ, id) -> acc@[typ]) [] args in
          let lfuns = SymTbl.add str (t1, args_types) lfuns in
```

```
            let lvars_new = List.fold_left (fun scope' (typ, var) -> (SymTbl.add var typ scope')) SymTbl.empty args in
            let env = [(lvars_new, SymTbl.empty); (lvars, lfuns)]@globals in (
          Sast.FDclr(t1, str, args, expr env e), env))
  in

  let emptyScope = [(SymTbl.empty, SymTbl.empty)] in

  fst (List.fold_left (fun (prog, env) st -> let stmt' = (stmt env st) in prog@[(fst stmt')], snd(stmt')) ([], emptyScope) ast)
```

(***)
(* src/sast.mli: SAST definition. Uses same types and operators as AST. *)
(***)

```
open Type
open Operator

type expr_detail =
  | LInt of int
  | LFloat of float
  | LList of expr list
  | LVctr of expr list
  | LTple of expr list
  | LString of string
  | Uop of uop * expr
  | Id of string
  | Binop of expr * binop * expr
  | Call of string * (expr * tp) list
  | Elmt of string * expr
  | If of expr * expr * expr
  | Scope of stmt list * expr
  (*| Noexpr *)

and expr = expr_detail * tp

and stmt =
  | FDclr of tp * string * (tp * string) list * expr
  | VDclr of tp * string * expr

and program = stmt list
```

(***)
(* src/compile.ml: Translator. Takes SAST and genrates OCaml code. *)
(***)

```
open Sast
open Type
open Operator

exception Bug of string (* For "impossible" situations *)

(* Main entry point: run a program *)
let translate prog out =
  (* Printing functions *)
  let put str = output_string out (str ^ " ") in
  let typestr = function
    | Float -> "float"
```

```
  | Int -> "integer"
  | Vector _ -> "vector"
  | RInt -> "randInt"
  | RFloat -> "randFloat"
  | List -> "list"
  | Str -> "string"
  | Tuple -> "tuple"
  | Func -> "function"
in

(* Conversion functions: wrap converters around expressions *)
let rec extotp ex = function
  | Float -> extofloat ex
  | Int -> extoint ex
  | Str -> extostring ex
  | _ -> eval ex

and extoint ex =
  let _, tp = ex in
  match tp with
  | Int -> eval ex
  | Float -> put "(int_of_float"; eval ex; put ")"
  | RInt -> defrand ex
  | RFloat -> put "(int_of_float "; defrand ex; put ")"
  | _ -> raise (Bug ("Tried to convert "^(typestr tp)^" to integer"))

and extofloat ex =
  let _, tp = ex in
  match tp with
  | Int -> put "(float_of_int"; eval ex; put ")"
  | Float -> eval ex
  | RInt -> put "(float_of_int "; defrand ex; put ")"
  | RFloat -> defrand ex
  | _ -> raise (Bug ("Tried to convert "^(typestr tp)^" to float"))

and extostring ex =
  let _, tp = ex in
  match tp with
  | Str -> eval ex
  | Int -> put "(string_of_int"; eval ex; put ")"
  | Float -> put "(string_of_float"; eval ex; put ")"
  | RInt -> put "(string_of_int "; defrand ex; put ")"
  | RFloat -> put "(string_of_float "; defrand ex; put ")"
  | Vector _ -> (
      put "(let arr ="; eval ex;
      put ("in (\"<\" ^ string_of_float arr.(0) ^ " ^
          "(Array.fold_left (fun o f->(o^\",\"^(string_of_float f)))" ^
          "\"\" (Array.sub arr 1 (Array.length arr - 1)))^\">\"))")
      )
  | _ -> raise (Bug ("Tried to convert "^(typestr tp)^" to string"))

and defrand ex =
  let _, tp = ex in
  match tp with
  | RInt -> put "(Rand.getRandInt "; eval ex; put ")"
  | RFloat -> put "(Rand.getRandFloat "; eval ex; put ")"
  | _ -> raise (Bug ("Tried to define a " ^ (typestr tp)))
```

```
(* Evaluate an expression *)
and eval = function

   (* Literals *)
   | LInt(i), _ -> put (string_of_int i)
   | LFloat(f), _ -> put (string_of_float f)
   | LList(l), tp -> (
      put "([";
      List.iter (fun ex -> (eval ex; put ";")) l;
      put "])")
   | LVctr(l), _ -> ( (* Vectors are arrays of floats *)
      put "([|";
      List.iter (fun ex -> extofloat ex; put ";") l;
      put "|])")
   | LTple(l), _ -> (* Tuples are tuples: have at least 2 elements *)
      (match l with
      | [] -> raise (Bug "Empty tuple")
      | hd::tl ->
         put "(";
         eval hd;
         List.iter ( fun ex -> put ","; eval ex) tl;
         put ")")
   | LString(s), _ -> put ("\"" ^ s ^ "\"")

   (* Identifiers.  Original names preserved with single quote *)
   | Id(var), _ -> put (var ^ "'")

   (* Array indices *)
   | Elmt(arr, pos), _ -> (
      put (arr ^ "'.(");
      let _, tp = pos in
      (match tp with
      | Int -> eval pos;
      | Float -> (
         put "int_of_float ";
         eval pos)
      | _ -> raise (Bug "Non-scalar array index"));
      put ")")

   (* Unary operators *)
   | Uop(op, ex), tp -> (
      put "(";
      (match op with
      | Minus -> (
            match tp with
            | Float -> (put "-."; extofloat ex)
            | Int -> (put "-"; extoint ex)

            (* Create a fresh array with inverted elements *)
            | Vector n -> (
               (* We need to evaluate the vector first, then create
                * the inverted copy based off it *)
               put "let arr = (";
               eval ex;
               put (") in Array.init " ^ string_of_int n
                  ^ " (fun i -> -. (arr.(i)))"))
            | _ -> raise (Bug "Unary '-' expression is non-numeric")
            )
```

```
  | Not -> (
     put "if ";
     let _, stp = ex in
     (match stp with
     | Int -> eval ex
     | Float -> ( put "(int_of_float"; eval ex; put ")")
     | _ -> raise (Bug "Unary '!' applied to non-scalar"));
     put "=0 then 1 else 0")
  );
  put ")")

(* Binary operators.  This is big and (mostly) boring. *)
| Binop(e1, op, e2), tp -> (
  let _, tp1 = e1 in
  put "(";
  (match tp with

  (* Integer result *)
  | Int -> (
     match op with

     (* Arithmetic *)
     | Add -> (extoint e1; put "+"; extoint e2)
     | Sub -> (extoint e1; put "-"; extoint e2)
     | Mult -> (extoint e1; put "*"; extoint e2)
     | Div -> (extoint e1; put "/"; extoint e2)
     | Rmndr -> (extoint e1; put "mod"; extoint e2)

     (* Comparison *)
     | Equal -> (put "if"; extofloat e1; put "="; extofloat e2; put "then 1 else 0")
     | Neq -> (put "if"; extofloat e1; put "<>"; extofloat e2; put "then 1 else 0")
     | Less -> (put "if"; extofloat e1; put "<"; extofloat e2; put "then 1 else 0")
     | Leq -> (put "if"; extofloat e1; put "<="; extofloat e2; put "then 1 else 0")
     | Greater -> (put "if"; extofloat e1; put ">"; extofloat e2; put "then 1 else 0")
     | Geq -> (put "if"; extofloat e1; put ">="; extofloat e2; put "then 1 else 0")

     (* Logical *)
     | Land -> (put "if ("; extoint e1; put "<> 0) && (";
        extoint e2; put "<> 0) then 1 else 0")
     | Lor -> (put "if ("; extoint e1; put "= 0) && (";
        extoint e2; put "= 0) then 0 else 1")

     | _ -> raise (Bug "Non-integer operator has integer type")
     )

  (* Float result *)
  | Float -> (
     match op with

     (* Arithmetic: 2 scalar operands *)
     | Add -> (extofloat e1; put "+."; extofloat e2)
     | Sub -> (extofloat e1; put "-."; extofloat e2)
     | Mult -> (extofloat e1; put "*."; extofloat e2)
     | Div -> (extofloat e1; put "/."; extofloat e2)

     (* Dot product *)
     | Innrp -> (
        put "\nlet arr1 ="; eval e1; put "in";
```

```
      put "\nlet arr2 ="; eval e2; put "in";
      put "\nlet len1 = Array.length arr1 in";
      put ("\nlet rec dot r = function -1-> r " ^
      "| n-> dot (r+.arr1.(n)*.arr2.(n)) (n-1) in dot 0.0 (len1-1)")
      )

   | _ -> raise (Bug "Non-float operator has float type")
   )

(* Vector operators *)
| Vector _ -> (
   (* We'll evaluate the vectors first, then apply the operator to
    * the elements. *)
   (match tp1 with
   | Vector _ -> (put "\nlet arr1 = "; eval e1; put " in")
   | _ -> (put "\nlet flt1 = "; extofloat e1; put " in")
   );
   put "\nlet arr2 = "; eval e2; put " in";
   put "\nlet len1 = Array.length arr1 in";
   (* Check sizes? OCaml does it for me *)
   match op with

   (* Arithmetic: 2 vector operands *)
   | Add -> put "\nArray.init len1 (fun i-> arr1.(i)+.arr2.(i))"
   | Sub -> put "\nArray.init len1 (fun i-> arr1.(i)-.arr2.(i))"

   | Mult ->
      (match tp1 with
      (* Cross product.  Only valid for 3 dimension vectors *)
      | Vector _ ->
         put ("\n[| arr1.(1)*.arr2.(2) -. arr1.(2)*.arr2.(1) ; "
         ^ "arr1.(2)*.arr2.(0) -. arr1.(0)*.arr2.(2) ; "
         ^ "arr1.(0)*.arr2.(1) -. arr1.(1)*.arr2.(0) |]")

      (* Vector Scaling, or 1x1 * 1xn matrix multiplication *)
      | _ -> put "\nArray.init len1 (fun i-> flt1*.arr2.(i))"
      )

   | _ -> raise (Bug "Non-vector operator has vector type")
   )

(* List operators *)
| List -> failwith "Lists not yet implemented"

(* String operators *)
| Str -> (
   match op with

   (* Concatenation *)
   | Add -> ( extostring e1; put "^"; extostring e2 )

   | _ -> raise (Bug "Non-string operator has string type")
   )

| _ -> raise (Bug ("Binary operation on " ^ typestr tp))
);
put ")"
)
```

```
    (* Function calls *)
    | Call(f, actuals), tp -> (
                                    let fname =
                                            if Builtin.exists f then
                                                    Builtin.get_name f
                                            else
                                                    (f ^ "'")
                                    in
      put ("(" ^ fname);
      (match actuals with
      | [] -> ()
      | (ex,tp)::tl ->
            put "("; extotp ex tp;
            List.iter (fun (ex,tp) -> put ",";
extotp ex tp) tl;
            put ")");
      put ")")

    (* If expression *)
    | If(cond, truebody, falsebody), tp -> (
      put "(if";
      extoint cond;
      put "= 0 then (\n";
      extotp falsebody tp;
      put "\n) else (\n";
      extotp truebody tp;
      put "\n))")

    (* With expression *)
    | Scope(init, body), tp -> ( (* first run a declaration block, then body *)
      put "(";
      List.iter (fun stmt -> (dclr stmt; put "in\n")) init;
      extotp body tp;
      put ")")

  (* Print declarations *)
  and dclr = function
    | FDclr(tp, lname, formals, body) -> (
      put ("let rec " ^ lname ^ "'");
      (match formals with
      | [] -> ()
      | (_,name)::tl ->
            put ("("^name^"'");
            List.iter (fun (_,name) -> put ",";
            put (name^"'")) tl;put ")");
      put "=";
      extotp body tp)
    | VDclr(tp, lname, actual) -> (
      put ("let " ^ lname ^ "' =");
      extotp actual tp)
  in

  (* Kickstart translation with a statement block, and keep an eye out for
   * "begin" (we need its type and its arguments'). *)
  let rec kick = function
  | [] -> raise (Bug "No begin function")
  | stmt::tl -> (
```

```
      dclr stmt;
      put ";;\n";
      (match stmt with
      | FDclr(beg_tp, name, arg_ls, _) ->
            if name = "begin" then
               (* Anything after begin can't be reached, anyway *)
               beg_tp, List.fold_left (fun l (t,_)->t::l) [] arg_ls
            else
               kick tl
      | _ -> kick tl)
   )
   in kick prog
```

```
(******************************************************************************)
(* src/mcslc.ml:  Compiler entry point.  Calls other layers in sequence.      *)
(******************************************************************************)
try
   (* get input file and program arguments *)
   let cmd, file = match Array.to_list Sys.argv with
   [] -> failwith ("Check for sanity")
   | _::[] -> failwith ("No input file")
   | c::f::_ -> c, f
   in

   (* open code file *)
   let input =
      try open_in file with
      Sys_error(_) -> failwith ("Couldn't open file: " ^ file)
   in

   (* create output buffer *)
   let output =
      let outname = (
         let base = Filename.basename file in
         try Filename.chop_extension base with
         Invalid_argument(_) -> base ) ^ ".ml"
      in
      try open_out outname with
      Sys_error(_) ->
         failwith ("Couldn't open output file: " ^ outname)
   in

   (* scan, parse, "compile" *)
   let lexbuf = Lexing.from_channel input in
   let ast = Parser.program Scanner.token lexbuf in
   let sast = Check.chk ast in
   let ret_tp, param_tp = Compile.translate sast output in
   Main.prt output ret_tp param_tp
with
Failure(str) -> print_endline ("Error: " ^ str); exit 1
```

```
(******************************************************************************)
(* src/main.ml: After translation, adds code to control program execution.    *)
(******************************************************************************)
open Sast
open Type
```

```
(* print out the kickoff program *)
let prt output rtype ptypel =
   (* formatting for debug *)
   let put str = output_string output (str^"\n") in
   let _ = put "" in
   let pcount = List.length ptypel in

   (* convert parameters *)
   let rec get_val i = function
      | [] -> ()
      | tp::tl ->
            let var = ("_param" ^ (string_of_int i)) in
            let _ = match tp with
            | Float -> put ("let "^var^" = float_of_string " ^
                                                "Sys.argv.(" ^ (string_of_int i) ^ ");;")
            | Int -> put ("let "^var^" = int_of_string " ^
                                                "Sys.argv.(" ^ (string_of_int i) ^ ");;")
            (* TODO *)
            | Vector _
            | Tuple
            | RInt
            | RFloat
            | List -> failwith "Unimplemented begin argument"
            | Str -> put ("let "^var^" = " ^
                                                "Sys.argv.(" ^ (string_of_int i) ^ ");;")
            | Func -> failwith "Function type argument in begin"
            in
            get_val (i+1) tl
   in
   (* print begin' parameters *)
(*    let rec begin_arg i =
      if i < pcount
      then (
         put ("_param" ^ (string_of_int i) ^ ",");
         begin_arg (i+1))
      else if i == pcount
      then (
         put ("_param" ^ (string_of_int i) );
         begin_arg (i+1))
      else ()
*)
   let rec begin_arg i =
      if i < pcount
      then (
         if(i == 1) then put("(");
         put ("_param" ^ (string_of_int i) ^ ",");
         begin_arg (i+1))
      else if i == pcount
      then (
         if(i == 1) then put("(");
         put ("_param" ^ (string_of_int i) ^ ")");
         begin_arg (i+1))
      else ()

   in (
      get_val 1 ptypel;
      put "let _ret = begin'";
```

```
    begin_arg 1;
    put " in print_endline";
    (match rtype with
    | Int -> put "(string_of_int _ret)"
    | Float -> put "(string_of_float _ret)"
    | Vector _ -> put ("(\"<\" ^ string_of_float _ret.(0) ^ " ^
          "(Array.fold_left (fun o f->(o^\",\"^(string_of_float f)))" ^
          "\"\" (Array.sub _ret 1 (Array.length _ret - 1)))^\">\")")
    | Str -> put "_ret"
    (* TODO *)
    | Tuple
    | RInt
    | RFloat
    | List -> failwith "Unimplemented begin return"
    | Func -> failwith "Function type return for begin"
    );
    exit 0
  )
```

```
(*******************************************************************************)
(* src/builtin.ml: Definition of builtin functions.                          *)
(*******************************************************************************)
open Type

module FMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

let ls = FMap.empty

(* Builtin Function Declaration Format:
 *   let ls = FMap.add MCSL_fname (MCSL_ret_tp, MCSL_arg_tp, OCaml_fname) ls
 * Where:
 *   @MCSL_fname  (string):  MCSL name of builtin function.
 *   @MCSL_ret_tp (tp):      MCSL return type.
 *   @MCSL_arg_tp (tp list): List of MCSL argument types.
 *   @OCaml_fname (strung):  Name of Ocaml function to call.
 *
 * Basically, the MCSL function call will get replaced with call
 * to the OCaml equivalent.  The evaluated arguments will be passed.
 * e.g. if the MCSL function was called with a Vector(3), the OCaml
 * function will be called with an Array of size 3.
 *)

(*** Begining of builtin fuction declarations ***)
let ls = FMap.add "VectorLength" (Float, [Vector 0], "Vector.length") ls
let ls = FMap.add "VectorDimension" (Int, [Vector 0], "Vector.dimension") ls
let ls = FMap.add "MCaggregate" (Float, [Func;Tuple;Int], "Mc.aggregate") ls  (* FIXME: correct types *)
let ls = FMap.add "MClist" (Float, [Str;Int], "Mc.list") ls  (* FIXME: correct types *)
let ls = FMap.add "MathFactorial" (Int, [Int], "Math.factorial") ls
let ls = FMap.add "MathAbs" (Int, [Int], "Math.abs") ls
let ls = FMap.add "MathFAbs" (Float, [Float], "Math.fabs") ls
let ls = FMap.add "MathPower" (Int, [Int;Int], "Math.pow") ls
let ls = FMap.add "RandFloat" (RFloat, [Float;Float], "Rand.floatRng") ls
let ls = FMap.add "RandInt" (RInt, [Int;Int], "Rand.intRng") ls
(*** End of builtin fuction declarations ***)
```

```
let exists fname = FMap.mem fname ls
let get_types fname = let ret_tp, arg_tp, _ = FMap.find fname ls in ret_tp, arg_tp
let get_ret_type fname = let ret_tp, _, _ = FMap.find fname ls in ret_tp
let get_arg_types fname = let _, arg_tp, _ = FMap.find fname ls in arg_tp
let get_name fname = let _, _, name = FMap.find fname ls in name
```

(***)
(* src/makefile: Ummm... the makefile.   *)
(***)

```
COMMOBJ=scanner.cmo parser.cmo
COMPOBJ=builtin.cmo check.cmo compile.cmo main.cmo
INTPOBJ=interpret.cmo
HEADERS=type.cmi operator.cmi sast.cmi ast.cmi parser.cmi
BUILDS=mcsli mcslc

all: mcslc

mcslc: mcslc.ml $(HEADERS) $(COMMOBJ) $(COMPOBJ)
	ocamlc -o $@ $(COMMOBJ) $(COMPOBJ) mcslc.ml

mcsli: mcsli.ml $(HEADERS) $(COMMOBJ) $(INTPOBJ)
	ocamlc -o $@ $(COMMOBJ) $(INTPOBJ) mcsli.ml

# Borland won't accept "%.cmo: %.ml" type targets, uses old fashioned suffix
# targets
.ml.cmo:
	ocamlc -c $<

.mli.cmi:
	ocamlc -c $<

.mly.mli:
	ocamlyacc $<

.mly.ml:
	ocamlyacc $<

.mll.ml:
	ocamllex $<

clean:
	-rm -f *.cm? $(BUILDS) parser.ml parser.mli scanner.ml

.SUFFIXES: .ml .mll .mly .mli .cmi .cmo
```

(***)
(* src/interpret.ml: Before the compiler, we wrote this basic interpreter.   *)
(***)

```
open Sast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)
```

```
exception ReturnException of int
exception Bug of string (* For "impossible" situations *)

(* Main entry point: run a program *)
let run prog args =
   (* Find and return symbols from scope *)
   let rec getvar id = function
              [] -> raise (Failure ("undeclared identifier " ^ id))
              | loc::scp ->
                 let (vars,funs) = loc in
                 if NameMap.mem id vars then
                          NameMap.find id vars
                 else
                          getvar id scp
   in
   let rec getfun id = function
              [] -> raise (Failure ("undefined function " ^ id))
              | loc::scp ->
                 let (vars,funs) = loc in
                 if NameMap.mem id funs then
                          NameMap.find id funs
                 else
                          getfun id scp
   in

   (* Evaluate an expression and return value *)
   let rec eval scope = function
     Literal(i) -> i
   | Noexpr -> 1 (* must be non-zero for the for loop predicate *)
   | Id(var) -> getvar var scope
   | Uop (op, e) ->
      let v = eval scope e in
      let boolean i = if i then 1 else 0 in
      (match uop with
        Not -> boolean(!v)
      | Minus -> -v
   | Binop(e1, op, e2) ->
              let v1 = eval scope e1 in
              let v2 = eval scope e2 in
              let boolean i = if i then 1 else 0 in
              (match op with
                Add -> v1 + v2
              | Sub -> v1 - v2
              | Mult -> v1 * v2
              | Div -> v1 / v2
              | Equal -> boolean (v1 = v2)
              | Neq -> boolean (v1 != v2)
              | Less -> boolean (v1 < v2)
              | Leq -> boolean (v1 <= v2)
              | Greater -> boolean (v1 > v2)
              | Geq -> boolean (v1 >= v2))
   | Call(f, actuals) ->
              let fdecl = getfun f scope in
              let actuals = List.fold_left
                 (fun actuals actual ->
                          let v = eval scope actual in v :: actuals) [] actuals
              in
              try call fdecl actuals scope
```

```
        with ReturnException(v) -> v

(* Invoke a function and return value *)
and call fdecl actuals globals =
        (* Enter the function: bind actual values to formal arguments *)
        let (fform, fbody) = fdecl in
        let lvars =
          try List.fold_left2
                (fun locals formal actual ->
                  let _, name = formal in
                  NameMap.add name actual locals)
                NameMap.empty fform actuals
          with Invalid_argument(_) ->
                raise (Failure ("wrong number of arguments passed to function"))
        in
        (* Execute function body, return returned value, ignore scope *)
        let ret, _ = exec ((lvars,NameMap.empty)::globals) fbody in ret

(* Run through a list of statements.  Return value of last statement and
 * modified scope.  This is used in enough places to justify a separate
 * function for it. *)
and stmtBlock scp statlist =
   List.fold_left (fun ret stmt ->
      let _ ,scp' = ret in exec scp' stmt) (0, scp) statlist

(* Execute a statement and return a value and updated scope *)
and exec scope = function
Block(stmts) ->
   let ret, _ = stmtBlock scope stmts in ret, scope
| Expr(e) -> eval scope e, scope
| FDclr(tp, lname, formals, body) -> (match scope with
   [] -> raise (Bug ("empty scope in FDclr"))
   | (lvars, lfuns)::globals ->
      let lfuns = NameMap.add lname (formals,body) lfuns in
      0, ((lvars, lfuns)::globals))
| VDclr(tp, lname, actual) -> (match scope with
   [] -> raise (Bug ("empty scope in VDclr"))
   | (lvars, lfuns)::globals ->
      let value, _ = exec scope actual in
      let lvars = NameMap.add lname value lvars in
      value, (lvars, lfuns)::globals)
| If(cond, truebody, falsebody) ->
      let ret, _ =
         if (eval scope cond) = 0
         then exec scope falsebody
         else exec scope truebody
      in ret, scope
| Scope(init, body) ->  (* first run a statement block, then body *)
      let _, scope' = stmtBlock scope init in
      let ret, _ = exec scope' body in
      ret, scope
| While(body, cond) ->
      let rec loop scp =
         let value, scp' = stmtBlock scp body in
         match eval scp' cond with
         0 -> value, scope
         | _ -> loop scp'
      in loop scope
```

```
   in

   (* Run a program: start with an empty scope and run through program.
    * Then find and call "begin", and print it's result *)
   let _, scope = stmtBlock [(NameMap.empty,NameMap.empty)] prog in
   try call (getfun "begin" scope) args scope
   with Failure(s) -> raise (Failure s)(*("did not find the begin() function"))*)
```

(**)
(* src/mcsli.ml: Entry point for interpreter. *)
(**)

```
let print = false in

(* get input file and program arguments *)
let cmd, file, args = match Array.to_list Sys.argv with
[] -> raise (Failure "Check for sanity")
| _::[] -> raise (Failure "No input file")
| c::f::[] -> c, f, []
| c::f::a -> c, f, List.fold_left (fun l e ->
      (int_of_string e)::l) [] a
in

(* open code file *)
let code = open_in file
   (*try open_in file with
   Sys_error -> raise (Failure "Couldn't open file: " ^ file)*)
in

(* scan, parse, interpret *)
let lexbuf = Lexing.from_channel code in
let program = Parser.program Scanner.token lexbuf in
if print then
   print_string "No printer yet"
else
   let ret = Interpret.run program args in
   print_endline (string_of_int ret)
```

(**)
(* lib/: Holds the source for the MCSL builtin functions. *)
(**)

(**)
(* lib/mc.ml: Monte Carlo algorithm functions. We intended to create variants. *)
(**)

```
let aggregate (func, args, times) =
   let rec aggregate_helper = function
      | 0 -> 0.0
      | 1 -> func args
      | n ->
         let p = n/2 in
         let q = n-p in
         (aggregate_helper p +. aggregate_helper q)
   in aggregate_helper times
;;
```

```
(* The reason why I do not use List.length is that it may take some time to execute, but I am not sure. *)
let list (func, args, times) =
  let rec list_helper n acc =
   if n >= times
     then acc
     else list_helper (n+1) ((func args)::acc)
  in
   list_helper 0 []
;;


(***********************************************************************************)
(* lib/math.ml: Main math library.                                                 *)
(***********************************************************************************)
(* This function does not check overflow. If argument is too large, this function will return 0.  *)
let rec factorial n =
  let rec factorial_helper x acc =
   if (x <= 1)
     then acc
     else (factorial_helper (x-1) (acc * x))
  in
  factorial_helper n 1
;;

let abs i = if i < 0 then -i else i ;;
let fabs i = if i < 0. then -.i else i ;;

let rec pow (b, e) =
  if e < 0 then
    failwith "pow: invalid argument"
  else
    let rec aux res = function
      | 0 -> res
      | n -> aux (res*b) (n-1)
    in aux 1 e



(***********************************************************************************)
(* lib/vector.ml: Vector operations.                                               *)
(***********************************************************************************)
let length arr =
        let sum = Array.fold_left
                (fun sum elem -> sum +. elem *. elem)
                0.0 arr
        in sqrt sum
;;

let dimension arr =
        Array.length arr


(***********************************************************************************)
(* lib/rand.ml: Random variable implementation and interface to GMP library.       *)
(***********************************************************************************)
open Gmp

type frng = FltRng of float * float;;
```

```
type irng = IntRng of int * int;;

let floatRng (a, b) = FltRng(a,b);;
let intRng (a, b) = IntRng(a,b);;
let randInit = RNG.default;;
let state = randInit;;
let getRandInt = function
      | IntRng(lo,hi) ->
            let boundary = hi-lo
            in
            let zboundary = Z.of_int boundary
            in
            let c = Z.urandomm state zboundary
            in
            let res = Z.to_int c
            in
            res+lo
let getRandFloat = function
      | FltRng(lo,hi) ->
            let precision = 7
            in
            let b = F.urandomb state precision
            in
            let res = F.to_float b
            in
            res*.(hi-.lo)+.lo
```

(**)
(* lib/makefile: Makefile for libmcsl.cma *)
(**)

```
OBJS=vector.cmo mc.cmo math.cmo rand.cmo
LIBS=libmcsl.cma

all: $(LIBS)

libmcsl.cma: $(OBJS)
	ocamlc -o $@ -a gmp.cma $(OBJS)

# Borland won't accept "%.cmo: %.ml" type targets, uses old fashioned suffix
# targets

rand.cmo: gmp.cmi rand.ml
	ocamlc -c rand.ml

.ml.cmo:
	ocamlc -c $<

.mli.cmi:
	ocamlc -c $<

.mly.mli:
	ocamlyacc $<

.mly.ml:
	ocamlyacc $<

.mll.ml:
```

```
	ocamllex $<

clean:
	-rm -f *.cmo *.cmi $(LIBS)

.SUFFIXES: .ml .mll .mly .mli .cmi .cmo
```

(**)
(* bin/: Executables were to go here. We ended with a single one.   *)
(**)

(**)
(* bin/mcsl: Bash script to translate and compile mcsl files into executables.   *)
(**)

```
#!/bin/bash

# MCSL compiling script.  Automates steps for converting a mcsl source file
# into an executable, or an in between state.

Usage () { cat; } <<doc
Usage: $CMD [options] filename
mcsl is a frontend to mcslc and mcsli, respectively the Monte Carlo Simulation
Language's compiler and interpreter.  By default, it will compile the input
file into an executable.  Use the options to change its behaviour.
Options:
   -C <file>          Use <file> as mcsl compiler
   -h                 Print usage and exit
   -o <file>          Place output into <file>
   -t                 Only translate to ocaml, don't compile
doc

Error () {
   rm -f $RMLIST &>/dev/null
   echo "$CMD: ${1:-"error"}" >&2
   exit ${2:-1}
}

# Get compiler command and directory
CMD=${0##*/}
DIR=${0%/*}

# Defaults
MCSLC="$DIR/../src/mcslc"
MCSLI="$DIR/../src/mcsli"
MCLIB="$DIR/../lib"
COMPILE=true
LIBS="libmcsl.cma"
RMLIST=""

# Minimal check
if [[ -z $1 ]] ;
then Error "no input files";
fi

# Scan arguments
while [[ -n $1 ]] ;
```

```
do
   case $1 in
   # Use an alternative compiler executable
            -C)
            if [[ -z $2 ]] ;
            then Error "no file for -C option"
            fi
            MCSLC=$2
            shift 2
            ;;

   # Print usage and exit
            -h)
            Usage
            exit 0
            ;;

   # Set an output file
            -o)
            if [[ -z $2 ]] ;
            then Error "no file for -o option"
            fi
            OUT=$2
            shift 2
            ;;

   # No compilation
            -t)
            COMPILE=
            shift
            ;;

   # Get input file and check for existance and extension
            *)
            if [[ -n $SRC ]] ;
            then Error "too many input files";
            fi

            SRC=$1
            if [[ ! -r $SRC ]] ;
            then Error "can't read file: $SRC";
            fi

            BASE=${1%.*}
            BASE=${BASE##*/}
            EXT=${1##*.}
            case $EXT in
               "mcsl")
               ;;
               *)
               Error "unknown filetype: $SRC"
               ;;
            esac
            shift
            ;;
   esac;
done
```

```
# Translate mcsl to ml
echo "Translating..."
if [[ ! -x $MCSLC ]] ;
then Error "can't execute compiler"
fi
RMLIST="$RMLIST $BASE.ml"
$MCSLC $SRC
if [[ $? -ne 0 ]] ;
then
   Error
fi
if [[ ! $COMPILE ]] ;
then
   if [[ -n $OUT ]] ;
   then mv $BASE.ml $OUT
   fi
   exit 0
fi

# Compile ml to executable
echo "Compiling..."
OCAMLC=$(which ocamlc)
if [[ $? -ne 0 ]] ;
then Error "can't find ocamlc"
fi
$OCAMLC -o ${OUT:=$BASE} -I $MCLIB $LIBS $BASE.ml
if [[ $? -ne 0 ]] ;
then Error
fi
RMLIST="$RMLIST $BASE.cmo $BASE.cmi"
rm -f $RMLIST &> /dev/null
```

(***)
(* tests/: Various test files and scripts to find bugs and show off.   *)
(***)

(***)
(* tests/test: Main test script. Compiles and runs test sources and checks output.*)
(***)

```
#!/bin/sh

MCSL="./mcsl"

# Set time limit for all operations
ulimit -t 30

globallog=testlog
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
   echo "Usage: testall.sh [options] [.mcsl files]"
   echo "-k    Keep intermediate files"
```

```
   echo "-h    Print this help"
   exit 1
}

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

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

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

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

   echo -n "$basename..."

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

   generatedfiles="${basename}.out" &&
   Run "$MCSL" $1 ">" ${basename}.out &&
   Compare ${basename}.out ${reffile}.out ${basename}.out.diff

   # Report the status and clean up the generated files

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

```
        echo "###### FAILED" 1>&2
        globalerror=$error
  fi
  rm -f ${basename}
}

Check2() {
  error=0
  basename=`echo $1 | sed 's/.*\\///
            s/.mcsl//'`
  echo -n "$basename..."

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

  Run "$MCSL" $1 &&

  # Report the status and clean up the generated files

  echo "OK" &&
  echo "###### SUCCESS" 1>&2
  rm -f ${basename}
}


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

shift `expr $OPTIND - 1`

echo
echo "======================================"
echo "basic tests"
echo "======================================"

files="basictests/*.mcsl"

for file in $files
do
        Check $file 2>> $globallog
done

echo
echo "======================================"
echo "random tests"
echo "======================================"
echo
files="randtests/*.mcsl"

for file in $files
```

```
do
	Check2 $file 2>> $globallog
done

exit $globalerror
```

(***)
(* tests/check: Translate a single file. *)
(***)

```
#! /bin/bash

./mcsl basictests/$1.mcsl -t
ocamlc -I ../lib/ $1.ml
```

(***)
(* tests/mcsl: Version of compiler script tweaked for testing purposes. *)
(***)

```
#!/bin/bash

# MCSL compiling script.  Automates steps for converting a mcsl source file
# into an executable, or an in between state.

Usage () { cat; } <<doc
Usage: $CMD [options] filename
mcsl is a frontend to mcslc and mcsli, respectively the Monte Carlo Simulation
Language's compiler and interpreter.  By default, it will compile the input
file into an executable.  Use the options to change its behaviour.
Options:
   -C <file>        Use <file> as mcsl compiler
   -h               Print usage and exit
   -o <file>        Place output into <file>
   -t               Only translate to ocaml, don't compile
doc

Error () {
   rm -f $RMLIST &>/dev/null
   echo "$CMD: ${1:-"error"}" >&2
   exit ${2:-1}
}

# Get compiler command and directory
CMD=${0##*/}
DIR=${0%/*}

# Defaults
MCSLC="$DIR/../src/mcslc"
MCSLI="$DIR/../src/mcsli"
MCLIB="$DIR/../lib"
COMPILE=true
LIBS="libmcsl.cma"
RMLIST=""

# Minimal check
if [[ -z $1 ]] ;
then Error "no input files";
fi
```

```
# Scan arguments
while [[ -n $1 ]] ;
do
   case $1 in
   # Use an alternative compiler executable
             -C)
             if [[ -z $2 ]] ;
             then Error "no file for -C option"
             fi
             MCSLC=$2
             shift 2
             ;;

   # Print usage and exit
             -h)
             Usage
             exit 0
             ;;

   # Set an output file
             -o)
             if [[ -z $2 ]] ;
             then Error "no file for -o option"
             fi
             OUT=$2
             shift
 2
             ;;

   # No compilation
             -t)
             COMPILE=
             shift
             ;;

   # Get input file and check for existance and extension
             *)
             if [[ -n $SRC ]] ;
             then Error "too many input files";
             fi

             SRC=$1
             if [[ ! -r $SRC ]] ;
             then Error "can't read file: $SRC";
             fi

             BASE=${1%.*}
             BASE=${BASE##*/}
             EXT=${1##*.}
             case $EXT in
                "mcsl")
                ;;
                *)
                Error "unknown filetype: $SRC"
                ;;
             esac
             shift
```

```
        ;;
    esac;
done

# Translate mcsl to ml
if [[ ! -x $MCSLC ]] ;
then Error "can't execute compiler"
fi
RMLIST="$RMLIST $BASE.ml"
$MCSLC $SRC
if [[ $? -ne 0 ]] ;
then
    Error
fi
if [[ ! $COMPILE ]] ;
then
    if [[ -n $OUT ]] ;
    then mv $BASE.ml $OUT
    fi
    exit 0
fi

# Compile ml to executable
OCAMLC=$(which ocamlc)
if [[ $? -ne 0 ]] ;
then Error "can't find ocamlc"
fi
$OCAMLC -o ${OUT:=$BASE} -I $MCLIB $LIBS $BASE.ml
if [[ $? -ne 0 ]] ;
then Error
fi
RMLIST="$RMLIST $BASE.cmo $BASE.cmi"
rm -f $RMLIST &> /dev/null
./$OUT
```

##basictests/addf.mcsl

```
float begin():=
10.345+2
```

##basictests/addf.out

```
12.345
```

##basictests/addi.mcsl

```
int begin():=
10+2
```

##basictests/addi.out

```
12
```

##basictests/addstring.mcsl

```
string begin():=
32+"hihi"
```

##basictests/addstring.out

```
32hihi
```

##basictests/addvec.mcsl
```
vector begin():=
with
		vector a:= <<1, 2>>
		vector b:= <<2, 3>>
do
		a+b
done
```

##basictests/addvec.out
```
<3.,5.>
```

##basictests/aggregate.mcsl
```
float twice (float x, float y) := 2 * x
float begin() := MCaggregate (twice, (1.0, 2.0), 100)
```

##basictests/aggregate.out
```
200
```

##basictests/and.mcsl
```
int begin():=
1&0
```

##basictests/and.out
```
0
```

##basictests/answer.mcsl
```
int begin() := 42
```

##basictests/answer.out
```
42
```

##basictests/arithvec.mcsl
```
vector begin():=
with
		vector a:= <<1, 2, 3>>
		vector b:= <<2, 3, 4>>
		vector c:= <<1, 3, 4>>
		vector d:= c
do
		a-b+d
done
```

##basictests/arithvec.out
```
<0.,2.,3.>
```

##basictests/comment.mcsl
```
/* gdsagdsgsdagdsagdsgdsgsdagsdgsdagasdgdsagsdagsdagsdgsdagasdgasdg*/
int begin():=
    /* aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa */
    0
```

```
/*dsakjfkdsjkkdsjgiujewohrjweqirnwmqrcowy4ryn4idskjferyr84yncifhyvksdlks*/
```

##basictests/comment.out

```
0
```

##basictests/crossvec.mcsl

```
vector begin():=
with
	vector a:= <<1, 2, 3>>
	vector b:= <<2, 3, 4>>
do
	a*b
done
```

##basictests/crossvec.out

```
<-1.,2.,-1.>
```

##basictests/dec.mcsl

```
int begin():=
with
	int a := 3
	int a := a+3
do
	a
done
```

##basictests/dec.out

```
666666
```

##basictests/dividef.mcsl

```
float begin():=
55.55/5
```

##basictests/dividef.out

```
11.11
```

##basictests/dividei.mcsl

```
int begin():=
50/5
```

##basictests/dividei.out

```
10
```

##basictests/fundec.mcsl

```
float func(float a):=
a*2

float begin():=
with
	float a := 12.12
do
	func(a)
```

done

##basictests/fundec.out
24.24

##basictests/if.mcsl
```
float begin() :=
with
    float a := 3.5
do
        if a < 1
                then 1
        else
                a
        endif
done
```

##basictests/if.out
3.5

##basictests/ifelse.mcsl
```
int begin() :=
with
        int a := 50
do
        if a < 1
                then 1
        elseif a > 100
                then 100
        elseif a == 50
                then 0
        else
                a
        endif
done
```

##basictests/ifelse.out
0

##basictests/mathabs.mcsl
```
int begin():=
MathAbs(-9)-MathAbs(9)
```

##basictests/mathabs.out
0

##basictests/mathfabs.mcsl
```
float begin():=
MathFabs(-1.3456)+MathFabs(1.1)
```

##basictests/mathfabs.out
2.4456

##basictests/mathfac.mcsl
```
int begin():=
	MathFactorial(6)
```

##basictests/mathfac.out
720

##basictests/mathpower.mcsl
```
int begin():=
MathPower(2,10)
```

##basictests/mathpower.out
1024

##basictests/minusf.mcsl
```
float begin():=
212-12.0001
```

##basictests/minusf.out
199.9999

##basictests/minusi.mcsl
```
int begin():=
212-12
```

##basictests/minusi.out
200

##basictests/mod.mcsl
```
int begin():=
4%3
```

##basictests/mod.out
1

##basictests/multiplef.mcsl
```
float begin():=3*22.22
```

##basictests/multiplef.out
66.66

##basictests/multiplei.mcsl
```
int begin():=
22*3
```

##basictests/multiplei.out
66

##basictests/nest.mcsl
```
int begin() :=
with
```

```
	int x := 3
do
	if x < 4
	then
		if
			with
				int y := 12
			do
				y/x
			done
		then  1
		else  -1
		endif
	else
		0
	endif
done
```

##basictests/nest.out

```
1
```

##basictests/or.mcsl

```
int begin():=
0|1
```

##basictests/or.out

```
1
```

##basictests/paren.mcsl

```
float begin() :=
(10 + 2*(3.4-2.4)+5)/2+5
```

##basictests/paren.out

```
13.5
```

##basictests/rec.mcsl

```
int longlong(int a):=
if a < 1
	then a
else
	longlong(a-1)
endif

int begin() :=
longlong(1000)
```

##basictests/rec.out

```
0
```

##basictests/scalarvec.mcsl

```
float begin():=
with
	vector a:= <<1, 2, 3>>
	vector b:= <<2, 3, 4>>
```

```
do
		a.b
done
```

##basictests/scalarvec.out

```
20.
```

##basictests/scope1.mcsl

```
int begin() :=
with
  int a := 3
do
		with
				int a:= 4
		do
				a*a
		done
		*a
done
```

##basictests/scope1.out

```
48
```

##basictests/scope2.mcsl

```
int dup():=
with
		int a:= 5
do
		a*a
done

int begin() :=
with
  int a := 3
do
		with
				int a:= 4
		do
				dup()
		done
		*a
done
```

##basictests/scope2.out

```
75
```

##basictests/string.mcsl

```
str begin():=
"hello word a  @@##%%^^&&()' t '"
```

##basictests/string.out

```
hello word a  @@##%%^^&&()' t '
```

##basictests/stringdec.mcsl

```
string s(string a, int b):=
a+b

string begin():=
with
        int a:= 21
        int b:= 12
do
        s(a,b)
done
```

##basictests/stringdec.out

```
2112
```

##basictests/vecdec.mcsl

```
int a:= 1
float b:= 2.2
vector v := <<a,b,3>>
vector begin() :=
   v
```

##basictests/vecdec.out

```
<1.,2.2,3.>
```

##basictests/vecdim.mcsl

```
vector v:= <<3,4>>

int begin():=
  VectorLength(v)
```

##basictests/vecdim.out

```
5
```

##basictests/veclength.mcsl

```
vector v:= <<1,2,3,4,5>>

int begin():=
  VectorDimension(v)
```

##basictests/veclength.out

```
5
```

##basictests/withdo.mcsl

```
float begin() :=
with
 int a := 3
 float b:= 4
do
 a*2+b
done
```

##basictests/withdo.out

```
10.
```

##randtests/addf.mcsl

```
float begin():=
with
	randFloat domain := RandFloat(0.0, 1.0)
do
	domain + 1.2
done
```

##randtests/addi.mcsl

```
int begin():=
with
	randInt domain := RandInt(0.0, 1.0)
do
	domain + 2
done
```

##randtests/arg.mcsl

```
randFloat domain := RandFloat(0.0, 1.0)
float s(randFloat f):=
f+1

float begin():=
	s(domain)
```

##randtests/arith.mcsl

```
float begin():=
with
	randFloat f := RandFloat(4.0, 20.0)
	randInt i := RandInt(4, 20)
	float s := f
do
	(f+i)*s
done
```

##randtests/dec.mcsl

```
randFloat domain := RandFloat(0.0, 1.0)

float begin():=
	domain
```

##randtests/randvec.mcsl

```
randFloat domain := RandFloat(0.0, 1.0)
vector s(randFloat f):=
<<f,f+1>>

vector begin():=
	s(domain)
```

##adtests/fac.mcsl

```
int isprime(int n):=
if n!=2 & n%2 == 0
```

```
        then 0
else
        with
                int checkprime(int n,int i):=
                        if i*i > n then 1
                        elseif n%i == 0 then 0
                        else  checkprime(n, i+2)
                        endif
        do
                checkprime(n, 3)
        done
endif

int gcd(int a, int b):=
if a == b
        then a
elseif a > b
        then gcd(a-b, b)
else
        gcd(b-a, a)
endif

int makeodd(int n):=
if n%2==0
        then makeodd(n/2)
else
        n
endif

string factorial(int n, int b, int k):=
with
        string str := ""
        int tmp := n-1
        randInt iran := RandInt(0,tmp-1)
        int a :=
                if iran <= 1
                then 2
                else iran
                endif
        int power := MathPower(a, k)
        int res := gcd(MathAbs((power)%n-1), n)
        int change := res > 1 & isprime(res)
        int n :=
                if change
                then n/res
                else n
                endif
        string str := if change then
                        str + " " +res
                else
                        str
                endif
do
        if isprime(n)
                then n+" "+str     /*print N */
        elseif n==1
                then str
        else
```

```
                    str+" "+factorial(n, b, k)
        endif
done

string fact(int n, int b):=
with
        int k := MathFactorial(b)
do
        factorial(n, b, k)
done


string begin(int n):=
with
        int b:= 6
        int n := makeodd(n)
do
        if n==1
        then ""
        elseif  isprime(n)
        then n
        else
     fact(n,b)
        endif
done
```

##adtests/pi.mcsl

```
float inCircle (randFloat x, randFloat y) :=
with
        vector v := <<x, y>>
do
        if VectorLength(v) <= 1
        then 1
        else 0
        endif
done

randFloat domain := RandFloat(0, 1)

float begin(int iterations) :=
        4 * (MCaggregate (inCircle, (domain, domain), iterations)) / iterations
```

##adtests/points.mcsl

```
randFloat domain := RandFloat(-100,100)
string point(randFloat rf) := <<rf,rf,rf>> + "\n"
string begin(int num) :=
  with
        string out := ""
        string aux(string str, int num) :=
          if num == 0 then
                    str
          else
                    aux (str + point(domain), num-1)
          endif
  do
        aux(out, num)
```

done