DaMPL Report

Programming Language & Translators

Bernardo Abreu
Felipe Rocha
Henrique Grando
Hugo Sousa

Spring 2016

Summary

Introduction	page 2
Language Tutorial	page 3
Project Plan	page 7
Translator Architecture	page 11
Test Plan	page 14
Lessons Learned	page 19
Language Reference Manual	appendix A
Code Listing	appendix B

Introduction

This report describes the experience of creating a new programming language and developing a full working compiler from scratch for the same language, as the main project in the class COMS W4115, Programming Languages & Translators, Spring 2016.

The language idealized is DaMPL (Data Manipulation Programming Language) and is meant to serve the purpose of facilitating the data manipulation, providing built-in types and operations that can easily handle and operate in large amounts of data in one step.

Among its features it is possible to highlight the CSV module developed in the DaMPL itself. It provides users with functions as readCSV that automatically stores a CSV file in a table (an array of tuples) that stores formatted data and allow further manipulation.

The main purpose as we built the compiler was to develop a fully working language, which could be used for the user to execute algorithms and perform data manipulation operations with many different file extensions. It is supposed to be a language focused on data manipulation, however, it is very flexible, in the sense that it is also very appropriate to be used in general purpose applications.

Language Tutorial

Type inference

In DaMPL, variables don't need to be declared. Once they're first instantiated, DaMPL automatically infers its type and keeps track of this information along the rest of the program.

```
inta = 0;  /* i inferred as integer */
stra = "";  /* str inferred as text */
numa = 1.2; /* num inferred as real */
```

Although the type is inferred, the variables have fixed types, that is, a variable's type cannot change.

```
inta = 0;       /* i inferred as integer */
inta = "a";     /* this line will cause an error */
```

Primitive Types

The primitive types in DaMPL are: integer, real, text and boolean.

```
inta = 0;              /* integer */
stra = "";             /* text */
numa = 1.2;            /* real */
booleana = false;      /* boolean */
```

The complex built-in types in DaMPL are: array and tuple.

Array

Arrays provide a lot of different manipulation features and exist to bring flexibility to the language.

```
arr = [];         /* Initializes empty array (type undefined) */
arr[] = 2;        /* Appends 2 to the end of array */
                  /* (and infers its type as array of integer) */
arr[0] = 1;       /* Set element at position 0 to be 1 */
@arr[0] = 0;      /*Insert an element at position 0 */
                  /* shifts remaining elements) */
arr[0];           /* Get element at position 0
```

```
arr2 = [3, 4, 5];     /* Initializes array of integers */
arr3 = arr + arr2;    /* Concatenates arr and arr2 */
arr3[0:4] = [3];      /* Sets the range [0,4) to be [3] */
arr1[:];              /* Gets the given range (whole array) */
```

Tuple

Tuples are used to store different types of data and to format data itself.

```
/* declares a tuple */
tuple Group {name, uni: integer, grade: real}
```

Tuples fields are inferred as text (name) by default, unless otherwise stated (uni and grade). The name of a tuple starts with a capital letter (Group).

Tuples fields can be accessed and changed in two different ways: index or name.

```
tup = Group;                 /* Initializes an empty tuple */
tup$(0) = "Stephen";         /* Sets field name using index */
tup$grade = 10.0;            /* Sets field grade using name */
```

When accessing or setting field by index, all values are considered as strings.

Table

Tables are itself the main application of tuples. Tables are nothing more than labeled arrays to store formatted data.

```
tab = Group[];               /* Initializes a table of Groups */
tab[] = tup;                 /* Appends tup to table */
```

All operations performed on arrays can also be performed on tables, however tables introduce an additional feature to assist in data manipulation.

```
arr = tab$uni;        /* Sets arr to be an array containing */
                      /* all UNIs in the given table */

tab$name = ["Dan"]    /* Sets all names in the table to the */
                      /* names contained in the array. */
           /* (If the sizes don't match, an error is thrown)*/
```

Control flow structures

DaMPL provides for loop, while loops and if statements.

```
for i in arr          /* for loop */
{
     print(i);        /* prints each element in the array */
}

while(true)           /* while loop */
{
     print("I'm dumb");
}

if (true)             /* if statement */
{
     print("I'm in");
}
```

Casting

It's possible to cast variables of one type to another

```
stra = str(9.3);      /* casts a real to text */

print(stra);          /* will print "9.3" */

print(int(stra));     /* will print 9 */
```

Function Calls and Declarations

To declare a function in DaMPL we use the keyword "fun".

```
fun sum (a,b)         /* Function declaration */
{
     return a+b;      /* Sum elements */
}

print (sum(9,3));          /* Prints 12 */
print (sum("h", "i"));     /* Prints "hi" */
```

The function above will work for different types of parameters, that is, if two integers are passed, it will perform an addition and return an int, if two strings are passed, it will perform a concatenation and return the resulting string.

Files

In DaMPL you can read/write files using single lines of code or open a file descriptor for a more complex handling of files.

```
str_file = readfile("input.txt");     /* Reads the whole file */
print(str_file);                      /* Print the string read */

/* Write string to file */
writefile("out.txt", "Bonjour Monde!");


file1 = open("text.txt", "r");        /* Open file descriptor */
bonjour = readline(file1);            /* Read line from file */

close(file1);                         /* Close file descriptor */
```

Project Plan

Planning, Specification, Development and Testing

Style Guide

Formating

Binary operators should have a space on either side. Commas and semicolons are always followed by whitespace. The keywords if, while, and for must be followed by a space. Parentheses should be used in expressions not only to specify order of precedence, but also to help simplify the expression. When in doubt, parenthesize.

Comments

The comments should be used to describe *what* the code does and how does it do it. For each function declared it should also describe what parameters mean, which globals are used and which are modified, and any restrictions or bugs. Comments that can be easily inferred from the code should be avoided.

Function Declarations

Each function should be preceded by a block comment describing its functionality, the parameters and the return variable, should it be present.

All function declarations should be done on the beginning of the file. In general the role of each variable in the function should be described. This may either be done in the function comment or, if each declaration is on its own line, in a comment on that line. Local variables that are used for common purposes, such as variables used in for loops, should have a name that indicate the function in which they are being used.

Comments for parameters and local variables should be tabbed so that they line up underneath each other.

Do not use local variable names that match the name of global variables. Functions that return a value should have at least one return statement that stands free on the function body.

Tuple Declarations

Tuples should always be declared on the beginning of the file. A comment may be included to describe the data being represented.

Coding

All variables should be initialized as soon as possible. All variable identifiers use letters ('A' through 'Z' and 'a' through 'z'), numbers ('0' through '9'), and underscores, and should begin with a lowercase letter.

Files Organization

The program files should start with all the necessary includes. The tuple and function declarations come next, and the the free code comes at the end.

Project Timeline

Roles of the Members

Bernardo: Language Guru

Felipe: Manager

Henrique: Compiler Architect

Hugo: Tester

Tools Used

Sublime Text: This text editor was used by all member of the group during development.

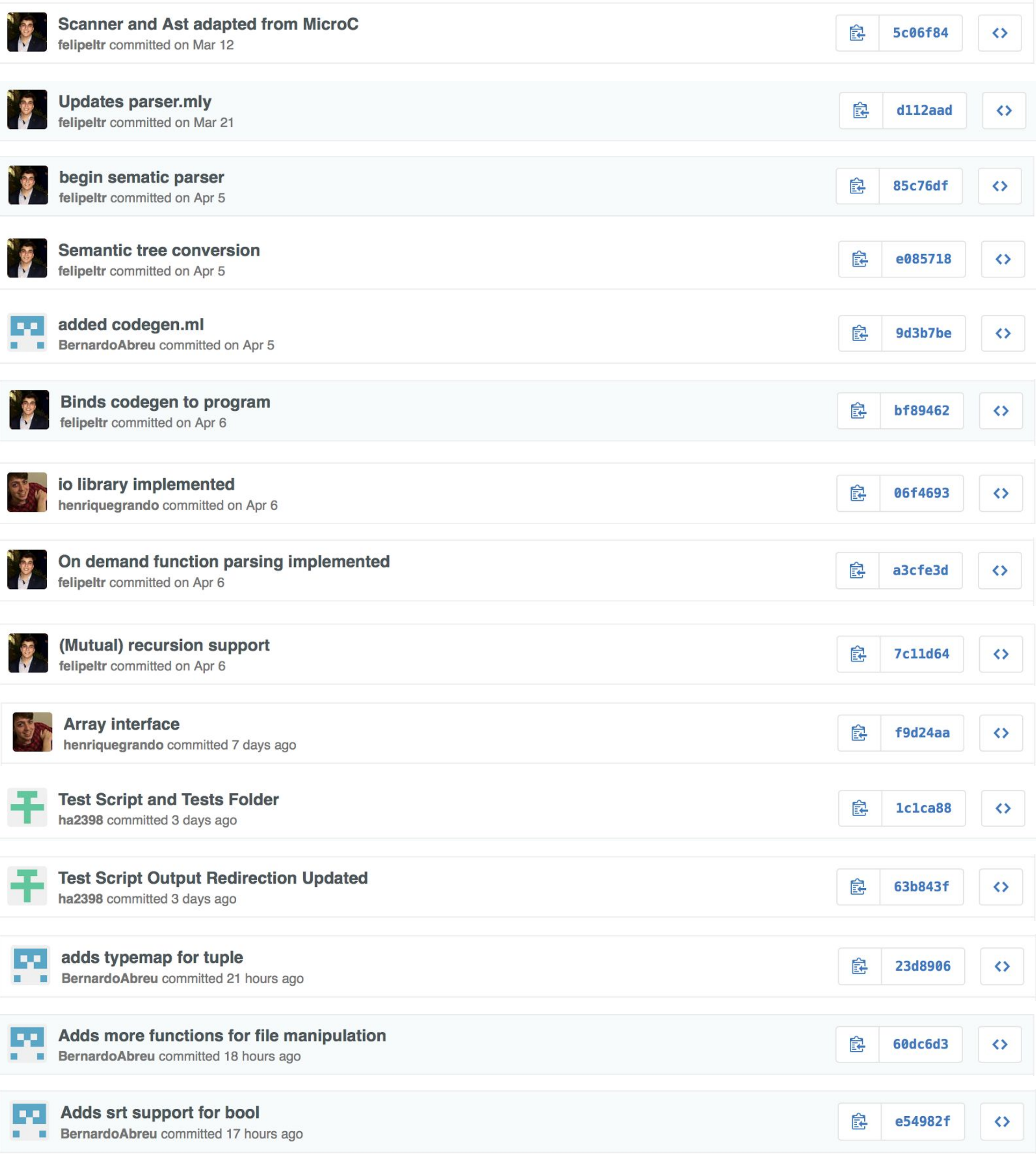
GCC: Since all code in DaMPL is translated to C, it`s necessary to have a C compiler in order to generate an executable file. Our choice was GCC. Hence, first the DaMPL code is fed to the DaMPL compiler and the output (a C source file) is fed to GCC, producing then the executable file.

UNIX: All code development was done in UNIX environment machines, two of the members used Mac OS and the other two used Ubuntu.

GitHub: GitHub was used both for versioning of code and as a secure way to store the code produced along the project.

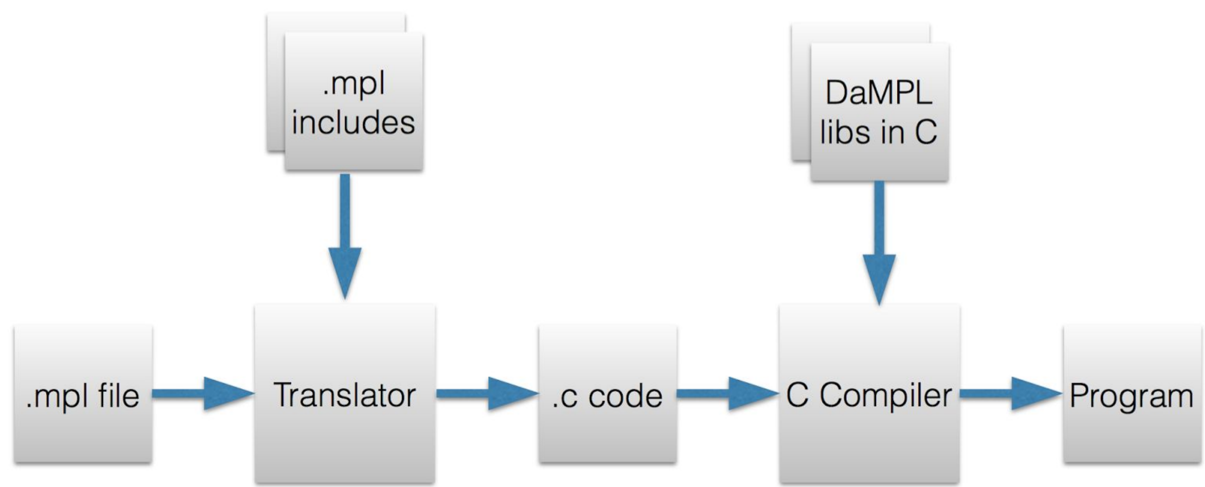
OCaml: The whole compiler was developed in OCaml and made use of features provided for this language such as ocamllex and ocamlyacc, used respectively for the scanner and parser.

Project Log



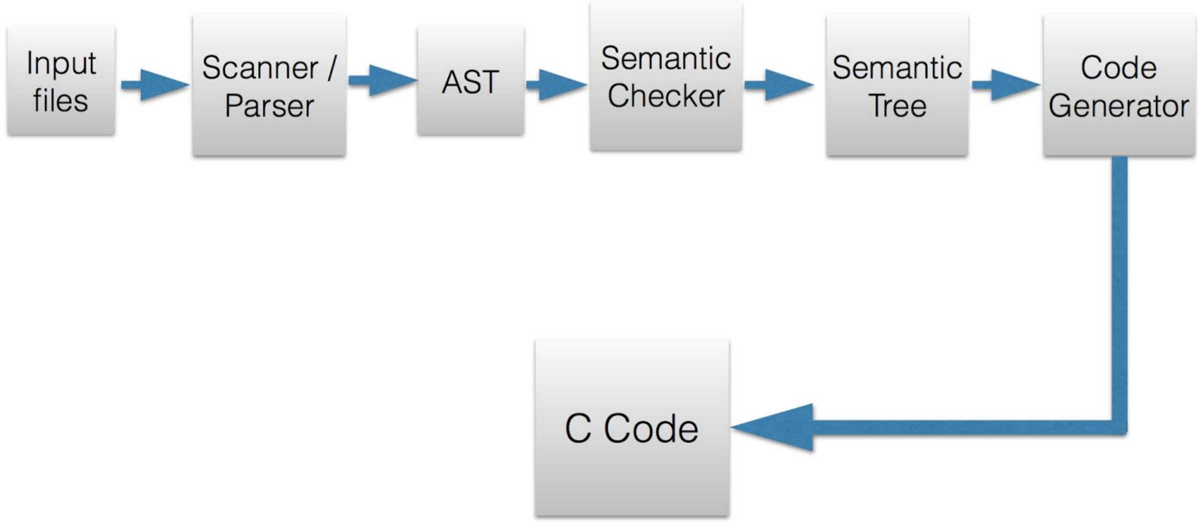
Translator Architecture

Program generation process:



The DaMPL translator takes .mpl files (DaMPL source code) and converts them into a single C code file. Built-in function, as well as array, table and tuple operations are implemented inside the DaMPL libs in C.

Translation process:



Input files: DaMPL code (main source file and includes)

Scanner/Parser: Composed by the scanner.mll (Ocamllex file), which converts words into tokens, and parser.mly (Ocamlyacc file), which takes the token and, using the specified grammar, builds the AST (Abstract Syntax Tree)

Semantic Checker: Written on sconv.ml file, it is responsible to infer types and check operations, building the Semantic Tree. The builtin.ml file is also used on this step to check for function calls that already exist on the DaMPL libs in C.

Code generation: Creates C code from the Semantic Tree.

Semantic Parsing

Since the language don't require that users explicit the variable type, the semantic checker must infer types for all variables and functions. So, types are inferred on variable assignments and functions can only be parsed when they are called, since this is the only way to determine parameters types.

Also, a function can also have different parameters types on different calls, which is handled as shown below:

DaMPL code

```
fun foo(p1,p2) {
    return p1+p2;
}

a=1;
b=1.2;
c=foo(a,b);

d="Hi ";
e="again";
f=foo(d,e);
```

C code

```
int dampl_a; float dampl_b; float dampl_c;
String dampl_d; String dampl_e; String dampl_f;

float dampl_foo__int_float
  (int dampl_p1,float dampl_p2) {
    return dampl_p1+dampl_p2;
}

String dampl_foo__str_str
  (String dampl_p1,String dampl_p2) {
    return dampl_str_concat(
           dampl_p1,dampl_p2);
}

int main() {
  dampl_a=1; dampl_b=1.2;
  dampl_c=dampl_foo__int_float(dampl_a,dampl_b);

  dampl_d="Hi "; dampl_e="again";
  dampl_f=dampl_foo__str_str(dampl_d,dampl_e);

  return 0;
}
```

In this case, the function could accept different parameters. So, for each call, a new prototype is built, if needed. Therefore, we use a parsing stack, which allows the checker to handle (mutual) recursion.

DaMPL code	Translate and check process

<pre>fun ping(a) {
    if(a>0) {
        print(“Ping... ”);
        pong(a);
    }
}
 	
fun pong(a) {
    print(“pong!\n”);
    ping(a-1);
}
 	
ping(3);</pre>
	<pre>First, build a function map with known functions
	including parameter count.
	-> [“ping”,1] [“pong”,1]
	
	Init stack with “_global_"
	Then, start reading statements
	-> ping(int) -> put “dampl_ping__int” on stack
	
	Start interpreting dampl_ping__int:
	-> if statement -> bool condition -> OK!
	-> print(str) -> use builtin “dampl_print__str”
	-> pong(int) -> put “dampl_pong__int” on stack
	
	Start interpreting dampl_pong__int:
	-> print(str) -> use builtin “dampl_print__str”
	-> ping(int) -> “dampl_ping__int” already on stack
	    \ -> ignore
	
	-> end of dampl_pong__int -> pop “dampl_pong__int”
	-> end of dampl_ping__int -> pop “dampl_ping__int"</pre>

A mutual recursion in which a function returns each other must be handled carefully. If this happen, then at the "ignore" step the caller function will be added to a list of functions to reparse. Therefore, the function is guaranteed to be fully parsed.

Test Plan

Automated Tests

Working with a version control system such as GitHub, it is desirable to test, for each change the code of the project goes through, if all the test files are still providing the expected output.

It is clear that, as the number of test files increase, the whole compile, test and evaluate process becomes impractical to be performed by hand, and especially test by test, manually.

This was the main motivation to incorporate **Regression Tests** in the project. That is, we need a tool to perform automatically all the necessary steps that are usually performed repeatedly to test our compiler and the test cases, which can be as many as desired.

For that purpose, a **bash script** was developed. This script has two main parts.

Build: This part of the script is responsible for compiling the source code for the DaMPL compiler and generating the executable **dampl**, which, given a **.mpl** (DaMPL source code extension) file, will compile it to a .c file.

Test: With the executable compiler in hands, we can test all the test files. For that, the script will look for all .mpl files in a particular test folder and, for each one of them, it will:

a) Compile from .mpl to .c using **dampl** compiler.
b) Compile the .c file generated on the last step and generate an executable file.
c) Run the executable file and store its output in a temporary file, **temp.out.**
d) Look for a file that corresponds to the expected output for this executable. This file must have the same name as the original .mpl file, but with extension .out.
e) Compare the output in temp.out with the expected output.
f) Store the differences in the two outputs in a **.diff** file.

This process will be repeated for all .mpl files (along with their correspondent .out files) in the test folder.

The script is also capable of detect compile time errors and then output the compiler's error message to the .temp.out file.For each file, the script will show a warning message if the expected output differs from the actual output for that file. Additionally, it will create a file called **files_to_check.txt,** which will contain the names of the files for which that happened.

Test files

With the script described in the previous section implemented properly, the process of testing the compiler consists in simply creating a new test file .mpl and a .out file with the expected output. Then, we add these two files to the test folder and run the script.

In total, 20 test files were developed to check the compiler's functionalities and proper programming. Their purpose is described below.
Obs.: For the code of the tests and their expected ouputs, .out files, see **Code Listing** section.

array_to_tuple.mpl: This file tests the feature of assigning an array to a tuple. This assignment needs to be done carefully, since the types will try to be assigned to the types in the tuple, being converted when necessary.

conflicting_types.mpl: Tests type inference as well as compatibility of types. This program is expected to fail at compile time.

fact.mpl: Tests recursion, and print function.

files.mpl: Tests the built in functions for file manipulation.

for_loop.mpl: Tests the for loop structure on arrays. Function tuple_to_array(t) and range(a,b) from **DaMPL_stdlib.mpl.**

helloworld.mpl: Classic Hello World! Program, tests printing a string.

index_out_of_bounds.mpl: Access an index out of bounds of an array. This program is expected to fail at running time.

invert.mpl: Function definition and call. While loop. Array appending. len(a) function.

join.mpl: String appending.

print.mpl: Print function for several different data types.

print_void.mpl: Tries to print a void type element. Expected to fail at compile time.

range.mpl: range(a,b) and range_s(a,b,c) functions, from DaMPL_stdlib.mpl.

str_concat: String concatenation and function call.

str_conversion.mpl: Tests conversion from several data types to text data type.

str_tuple.mpl: String and tuple manipulation.

test1.mpl, test2.mpl and test3.mpl: These files test the operation of including other files.

undefined_function.mpl: Tries to call a function that was never defined. Expected to fail at compile time.

undefined_variables.mpl: Similarly to the program above, this file tries to use variables that were never declared. Expected to fail at compile time.

Illustrative Examples

To make the whole process from building the compiler to evaluating the results of the generated executables, in these section there are two complete examples that illustrate what was described in the previous sections.

helloworld.mpl

This is one the most basic programs that can be written in DaMPL. It simply has a main function, which prints the string “Hello World!\n”. After building the DaMPL compiler, we can use it to generate a .c file using the file helloworld.mpl. The generated C code is shown below:

Code in DaMPL

```
fun main() {
   print("Hello World!\n");
}

main();
```

Code in C

```
#include <stdio.h>
#include <stdlib.h>
#include "dampllib.h"

Array  dampl_args;

void  dampl_main__();

void  dampl_main__()
```

```
{
dampl_print__str("Hello World!\n");
}

int main(int argc,char** argv){
dampl_args=build_args_array(argv);
dampl_file_constructor();
dampl_main__();
return 0;
}
```

Now the next step executed by the script is to compile the C code, using **gcc.** This will generate the executable file **helloworld.** Now it will run helloworld and obtain its output, which in this case is the string

Hello World!

This string will be compared to the content of **helloworld.out.** In this case, there is no difference between the output we obtained by running the executable and the .out file, so **helloworld.diff** will be an empty file.

index_out_of_bounds.mpl

This particular program has as its main purpose to test DaMPL's capacity to identify running time errors. Here, an array of integers will be created and the user tries to access a position of this array which is beyond its bounds. Note that this is not a compile time error, but as we try to run the program, it will generate the error.

Code in DaMPL

```
a = [1, 2, 3, 4, 5];

b = a[7];

print(b);
```

Code in C

```
#include <stdio.h>
#include <stdlib.h>
#include "dampllib.h"


int  dampl_b;
Array  dampl_args;
```

```
Array  dampl_a;


int main(int argc,char** argv){
dampl_args=build_args_array(argv);
dampl_file_constructor();
dampl_a = ({
Array a=dampl_arr_new();
dampl_arr_append__int(a,1);
dampl_arr_append__int(a,2);
dampl_arr_append__int(a,3);
dampl_arr_append__int(a,4);
dampl_arr_append__int(a,5);
a;});
dampl_b = dampl_arr_get__int(dampl_a,7);
dampl_print__int(dampl_b);
return 0;
}
```

After running gcc on the C source code, we obtain the **index_out_of_bounds** executable. Running this program, we expect the execution to be terminated on the line

b = a[7]

Since the array **a** has only size 6, hence, accessing position 7 will throw a "**Array out of bounds exception"** and the execution stops. We can see that this is the exact same message which is output to the program.

Lessons Learned

Bernardo

Working on this project helped me learn how to better work in a group, dividing the work properly among the group members and how to coordinate the work done by each one. I learned about the importance of thinking ahead on the project, and to not try to do a single part all at once. It is more helpful to implements new features along all the compilation process.

As an advice for future students, I would recommend to plan ahead on the language, and start early, implementing the whole process, even if it is very simple at the start.

Felipe

I believe this project allowed us to get more experience in teamwork, which showed itself much harder than just dividing tasks. As a manager, I was able to learn about project management and deadlines requirement.

Academically, the best thing I learned was functional programming. A functional code is such a powerful tool to solve a lot a problems, and it has also been widely used by companies nowadays.

Henrique

When working on projects of this magnitude many skills are developed throughout the process, moreover not only new skills are developed but also you learn how to interconnect your different abilities and work with other's strengths to accomplish something unachievable on your own.

Among the things learned it is worth highlighting the in-depth structure of a compiler, the scanner, the parser, the different decisions to be made when designing a language, the differences that they bring to the language itself and to the compiler, how to express your ideas in terms of a grammar, teamwork among others.

As a piece of advice, I think I could say (despite how cliche it may sound) that starting early is crucial in this project. It will make the difference between an excruciating workload and a project that has incremental manageable goals that develop toward a main objective.

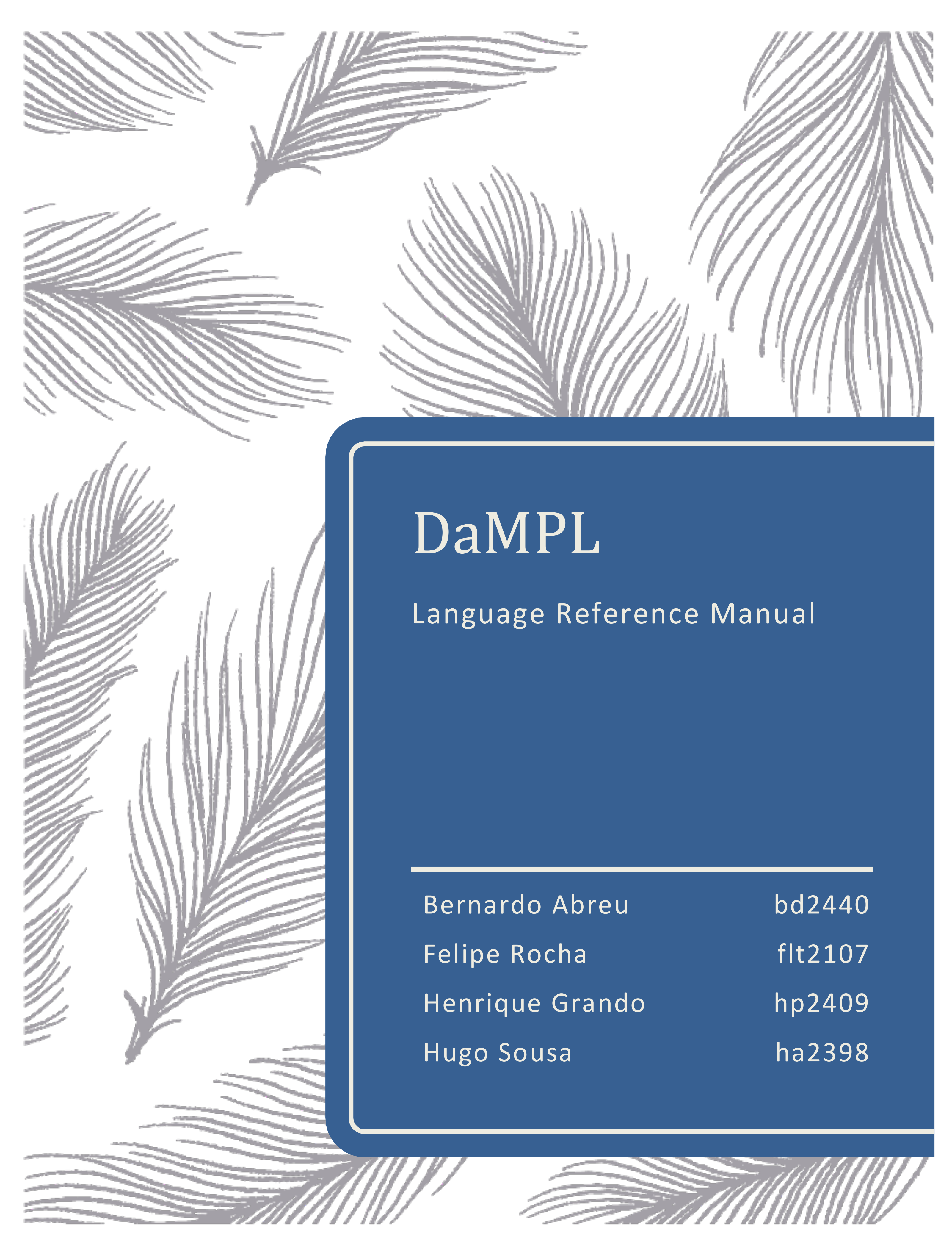
Hugo

Working on this project, I learned many important things. One of them, which I consider the most important, is how to efficiently work in a team. This does not only mean being able to split the chores, but to spread the workload in a timely manner, considering the amount of time we had (which was short, if we think about how many lines of code were written, for instance, or how complex a compiler is).

It was also very curious to see how, even with one position being assigned to each one of us, we were always cooperative and willing to help one another whatever the problem was. I was the tester, however, that does not mean simply creating a huge number of tests and expecting them to work, throwing the errors found to the other members to fix. It needs to be a cooperative work. And to develop important tests, the ones that are crucial to evaluating how well we are doing our jobs, it’s necessary to understand the languages’ features, what problems the group is facing or what are the ideas to implement next.

As for the next students, I would recommend them to start early, to spread the work along the semester, to have an efficient way to test the compiler from the beginning and to test it extensively.

Appendix A - Reference Manual


DaMPL
Language Reference Manual
Bernardo Abreu bd2440
Felipe Rocha flt2107
Henrique Grando hp2409
Hugo Sousa ha2398

Contents

1. Getting Started ... 4
2. Syntax Notations ... 4
3. Lexical Conventions ... 4
3.1. Line Structure ... 4
3.1.1. Logical Lines ... 4
3.1.2. Physical Lines ... 4
3.1.3. Comments ... 4
3.1.4. Blank Lines ... 4
3.1.5. Whitespace between tokens ... 5
3.2. Identifiers and Keywords ... 5
3.2.1. Identifiers ... 5
3.2.2. Keywords ... 5
3.3. Literals ... 5
3.3.1. String Literals ... 5
3.3.2. Integer Literals ... 5
3.3.3. Floating Point Literals ... 5
3.3.4. Boolean Literals ... 5
4. Statements ... 6
5. Expressions ... 6
5.1. primary-expression ... 6
5.2. Postfix-expression ... 6
5.2.1. Reference operator ... 7
5.3. unary-expression ... 7
5.3.1. not operator ... 7
5.3.2. - operator ... 7
5.4. Function calls ... 7
5.5. multiplicative-operators ... 7
5.5.1. * and / operators ... 8
5.6. additive-operators ... 8
5.6.1. + and – operators ... 8
5.7. Relational operators ... 8
5.7.1. <, >, <= and >= operators ... 8
5.8. Equality operator ... 9
5.8.1. == operator ... 9
5.9. AND operator ... 9
5.9.1. and operator ... 9
5.10. OR operator ... 9
5.10.1. or operator ... 9
5.11. Assignment operator ... 9
5.11.1. = operator ... 10
6. Variables ... 10
6.1. Types ... 10
6.2. Conversion ... 10
7. Tuples ... 10

8. Control Structures 11
8.1. If 11
8.2. While 12
8.3. For 12
9. Functions 12
9.1. Definitions 12
9.2. Calls 13
10. Arrays 13
11. Tables 14
12. Include 15
13. Program 15

1. Getting Started

DaMPL (Data Manipulation Programming Language) is a scripting language designed for high-level applications that require easy and robust data manipulation. Its features aim to define and manipulate information (either defined by the user or obtained from external sources) in a clear and concise way.

In this reference manual the reader is able to find a detailed description of the structure of DaMPL programs and explanations the main points of the language's features.

Within the language, there are many features to input and output differently structures data-types, as well as particular functions of the standard modules that manipulate specific file extensions, such as ".csv" files, which are well known and extensively used to store data.

DaMPL is implemented as translated to C (for more details, see the GNU C Reference Manual) language, using gcc. As a constantly evolving language, the implementation details are likely to change, therefore, the main focus on this manual is to provide a complete documentation.

2. Syntax Notations

For the next items of this manual, syntactic notations will be written in *italic,* and literal word and characters will be written as regular text. A definition of a new syntactic notation will be as follows:

notation-name:

<possibility-1>

...

<possibility-n>

3. Lexical Conventions

3.1. Line Structure

A program in DaMPL is divided into a number of logical lines.

3.1.1. Logical Lines

The end of a logical line is represented by the token SEMICOLON. A logical line is constructed from one or more physical lines.

3.1.2. Physical Lines

A physical line is a sequence of characters terminated by an end-of-line sequence.

3.1.3. Comments

There are one-line and multi-line comments in DaMPL. One-line comments start with a sequence of two bar characters (//).

Multi-line comments start with a sequence of a slash character followed by an asterisk character (/*) and end with a sequence of an asterisk character followed by a slash character (*/).

3.1.4. Blank Lines

A physical line consisting of only spaces, tabs, formfeeds and comments are ignored.

3.1.5. Whitespace between tokens

The whitespace characters space, tab and formfeed can be used interchangeably to separate tokens.

3.2. Identifiers and Keywords

3.2.1. Identifiers

An identifier in DaMPL can be defined as a sequence of letters and digits. It starts by a letter or underscore. Upper and Lower case letters are different.

For now on, identifier start with a lowercase letter and a tuple-identifier start with an uppercase letter.

3.2.2. Keywords

The following identifiers are keywords, or reserved words, of the language.

`and`	`for`	`include`	`tuple`
`break`	`fun`	`not`	`while`
`continue`	`if`	`or`	
`else`	`in`	`return`	

3.3. Literals

Literals are notations for representing constant values.

literal:

string-literal
integer-literal
floating-point-literal
boolean-literal

3.3.1. String Literals

A string literal is a sequence of characters surrounded by double quotes, as in " . . . ".

3.3.2. Integer Literals

A integer literal is a sequence of digits.

3.3.3. Floating Point Literals

A floating literal consists of an integer part, a decimal point, a fraction part, an e or 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/E and the exponent (not both) may be missing.

3.3.4. Boolean Literals

A boolean literal is described by the following definition.

boolean-literal:

true
false

4. Statements

statement:
expression;
control-structure
fun-def
tuple-definition

statements:
statement
statement statements

5. Expressions

The precedence of expression operators is the same as the order of the subsections within this section. Operators in the same subsection have the same precedence. The associativity (left or right) is specified for each operator within a subsection. An expression is defined as follows:

expression:
primary-expression
unary-expression
binary-expression
call-expression
multiplicative-expression
additive-expression
relational-expression
and-expression
or-expression
assignment-expression

5.1. primary-expression

A primary expression is a literal, an identifier or a parenthesized expression.

primary-expression:
literal
identifier
(expression)

5.2. Postfix-expression

The operators in a postfix expression are applied to an identifier and group left to right.

postfix-expression:

identifier
postfix-expression[expression]
postfix-expression[<empty>]
postfix-expression[expression:expression]
postfix-expression$identifier

5.2.1. Reference operator

Tables and Arrays elements can be accessed through a *postfix-expression[expression],* where *postfix-expression: identifier* for any of these types.

5.3. unary-expression

A unary expression group right to left.

unary-expression:
unop primary-expression

unop: one of
not -

5.3.1. not operator

The operand must have a boolean type. It will return the negated value.

5.3.2. - operator

The operand must have an integer type. Change the sign of the integer literal

5.4. Function calls

A function call is defined by a identifier (function name), followed by zero or more expressions as parameters.

call-expression:
identifier(*expression-list*)

expression-list:
expression
expression expression-list

5.5. multiplicative-operators

The multiplicative operators * and / group left-to-right.

multiplicative-expression:
expression
*multiplicative-expression * expression*
multiplicative-expression / expression
multiplicative-expression % expression

5.5.1. * and / operators

The operands for all these operators must have arithmetic type (int or float).

The * returns the multiplication result in the equivalent arithmetic type of the operands. In case of the operands being of different types, the int gets promoted to float.

The / returns the result of the division as a float. The right operand can't be zero.

5.6. additive-operators

The additive operators + and - group left-right.

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

5.6.1. + and – operators

The operands for these operators must have arithmetic type. In case the operands differ in type, the int is promoted to float.

The + returns the sum result in the equivalent arithmetic type of the operands.

The – returns the subtraction result in the equivalent arithmetic of the operands.

5.7. Relational operators

The relational operators group left-to-right.

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

5.7.1. <, >, <= and >= operators

The operands for these operators must have arithmetic type. In case the operands differ in type, the int is promoted to float.

The < operator returns true if the left operand is less than the right operand and false if not.

The > operator returns true if the left operand is greater than the right operand and false if not.

The <= operator returns true if the left operand is less than or equal to the right operand and false if not.

The >= operator returns true if the left operand is greater than or equal to the right operand and false if not.

5.8. Equality operator

Equality-expression:
  Relational-expression
  Equality-expression == relational-expression

5.8.1. == operator

The == operator tests if both operands are equal, i.e., have the same value.

5.9. AND operator

The && operator groups left-to-right.

and-expression:
  equality-expression
  and-expression && *equality-expression*

5.9.1. and operator

The operands for the and operator must be of type boolean.
The and operator returns a boolean value: true if both operands are true; false otherwise.

5.10. OR operator

The || operator groups left-to-right.

or-expression:
  and-expression
  or-expression || *and-expression*

5.10.1. or operator

The operands for the or operator must be of type boolean.
The or operator returns a boolean value: true if at least one of the operands is true; false otherwise.

5.11. Assignment operator

The = operator groups right-to-left.

assignment-expression:

postfix-expression = assignment-expression

5.11.1. = operator

The assignment operator expects the operands' types to be the same.
The assignment operator returns the right operand value.

6. Variables

DaMPL is not a typed language, i.e. you don't have to declare a variable along with its type, since it will be determined through type inference. To each variable is associated a value, an underlying type and an id (that is related to the position in memory where the variable data is stored).

Variable names (notation: *identifier*) must start with a lowercase letter or an underscore, and may contain letters, numbers and underscore.

6.1. Types

A type determines how the value stored in a variable is going to be interpreted. The types in DaMPL are the following: *int, float, str, bool.*

6.2. Conversion

Variables can be converted from one type to another using the constructors (notation: *call-expression)* offered for each type:

int (identifier)
float (identifier)
str (identifier)
bool (identifier)

7. Tuples

Tuples are associations of values, where each item has an label name. These are useful to deal with data rows as we'll see later on this manual.

A tuple name (notation: *tuple-identifier*) must always begin with an uppercase letter, and may contains characters or numbers.

Tuple labels (notation: *tuple-label*) must always begin with a lowercase letter, and may contain letters, numbers and underscore.

tuple-definition:
tuple *tuple-identifier*{ *tuple-label-list* }

tuple-instantiation:
tuple-identifier

tuple-item-identifier:

identifier$*tuple-label*

Note: The expression above is one of the postfix expressions repeated with different names for convenience

Where
tuple-label-list:
tuple-label
tuple-label, tuple-label-list

Note: *tuple-item-identifier* can be used also on the LHS of an assignment

Example

```
// This defines a new tuple name
tuple Person{name,age}

// This instantiates one variable as a tuple Person
a = Person;
a$name = "Steve";
a$age = 30;
```

8. Control Structures

iterable:
identifier
expression

Note: it must be a valid iterable (array or table)

iterator_variable:
identifier

8.1. If

If statements are used for conditional execution and are defined as follows:

if-else:
if *expression* { *statements* }
if *expression* { *statements* } else { *statements* }
if *expression* { *statements* } else *if-else*

It evaluates the expression, which must be of boolean-equivalent type, and executes the first group of statements if this binary-expression is true. If the binary-expression is false and the else exists, the statements or if-else that follows the else are executed.

8.2. While

While statements are used for repeated execution of a statement as long as a binary expression is true. They are defined as follows:

while:
    while *expression* { *loop-statements* }

loop-statements:
    statements
    break; *statements*
    continue; *statements*

This repeatedly tests the expression, which must be of boolean-equivalent type, and, if it is true, executes the loop-statements. If the expression is false, the loop terminates.

A break executed in the loop-statements terminates the loop, while a continue executed in it skips the rest of the statements and goes back to testing the expression.

8.3. For

For statements are used to iterate through a iterable. They are defined as follows:

for:
    for *iterator-variable* in *iterable* { *loop-statements* }

loop-statements:
    statements
    break; *statements*
    continue; *statements*

For each item provided by the iterable, this item is assigned to the iterator-variable using the standard rules for assignments, and then the loop-statements are executed. When the items are exhausted, which means that the for has gone through all the items on the iterable, the loop terminates.

A break statement executed in loop-statements terminates the loop, while a continue statement executed in it skips the rest of the suite and continues with the next item, or terminates the loop if there was no next item.

9. Functions

9.1. Definitions

DaMPL allows the user to define their own functions in order to manipulate data structures in the way that best fits their needs.

Function names (which also use notation *identifier*, just like variable names) must start with a lowercase character or an underscore, and may contain characters, numbers and underscore.

In order to start a new function definition, the user needs to use the keyword *fun,* as shown below:

fun-def:
fun identifier(parameter-def-list) { function-body }

Where
parameter-def-list:
identifier
identifier, parameter-def-list

And *function-body* is a sequence of logical lines, the last one being usually (but not mandatorily) a return statement.

function-body:
statements
return expression; statements

The return keyword tells the function what is the expression that will be evaluated and returned to the function caller.

Examples:

```
fun myFunction123(arg1, arg2) {
      return (arg1 + arg2);
}

fun anotherFunction1() {
      print("Hello World");
}
```

9.2. Calls

Function calls are already defined under the expressions section

10. Arrays

Arrays holds values of same-type (allowed internal types: integer, floating point, boolean or string), which can be accessed by its zero-indexed positions.

array-expression:
[*array-items*]

array-items:
<empty>
integer-items

floating-point-items
boolean-items
string-items
array-items

<x>-items:
<x>-literal
<x>-literal, <x>-items

Example:

```
a = [];
a[] = 1; // This adds 1 to the end of a
a[] = 2;
a[] = 3;
a[] = 4;

// The following would have the same result
b = [1,2,3,4];

b[0]; //equals 1
b[1:3]; // equals [2,3]
```

11. Tables

Tables holds same-tuple-label instances. To define a table the user needs to specify the tuple that defines the structure of the table.

table-instantiation:
tuple-identifier[]

table-indexing:
identifier$tuple-label

In order to add an element (tuple) to the table, we use brackets, as follows:

Example:

```
tuple Foo{fa,fb,fs}; //defines a tuple Foo

a = Foo; //Creates a tuple a
a$fa = 1;
a$fb = 2.2;
a$fs = "abc"

b = Foo;  //Creates a tuple b
```

```
b$fa = 2;
b$fb = 3.0;
b$fs = "def";


t = Foo[];          //Instantiates a table that stores Foo tuples

t[] = a;        //adds tuple a to table t
t[] = b;        //adds tuple b to table t

t$fa;    //returns all fa elements in the tuples stored ([1,2])

t[0];    //returns the first tuple of the table ({1,2,"abc"})
```

12. Include

A file of DaMPL code can gain access to another file with a include statement. This statement allows for the first file to access the content of the second one as if its code were present on the same file as the first.

include-statement:
include *string-expression*;

13. Program

program:
includes
statements

includes:
include
include includes

For a detailed description on the language grammar, check the parser.mly file inside the DaMPL official source code.

Appendix B - Code Listing

DaMPL Source Code

May 11, 2016

Contents

1 DaMPL_stdlib.mpl 5

2 helloworld.mpl 8

3 libs/damplarray.c 9

4 libs/damplarray.h 30

5 libs/damplio.c 33

6 libs/damplio.h 40

7 libs/dampllib.c 41

8 libs/dampllib.h 45

9 libs/damplstr.c 46

10 libs/damplstr.h 47

11 libs/dampltup.c 48

12 libs/dampltup.h 52

13 libs/Makefile 53

14 Makefile 54

15 mycsv.mpl 55

16 sandbox.mpl 56

17 src/ast.ml **57**

18 src/builtin.ml **59**

19 src/codegen.ml **61**

20 src/main.ml **67**

21 src/Makefile **68**

22 src/parser.mly **69**

23 src/scanner.mll **73**

24 src/sconv.ml **75**

25 src/semt.ml **93**

26 src/tokenize.sh **95**

27 src/tokenizer.ml **96**

28 testall.sh **99**

29 tests/array_to_tuple.mpl **101**

30 tests/array_to_tuple.out **102**

31 tests/conflicting_type.mpl **103**

32 tests/conflicting_type.out **104**

33 tests/fact.mpl **105**

34 tests/fact.out **106**

35 tests/files.mpl **107**

36 tests/files.out **108**

37 tests/for_loop.mpl **109**

38 tests/for_loop.out **110**

39 tests/helloworld.mpl **111**

40 tests/helloworld.out **112**

41 tests/index_out_of_bounds.mpl **113**

42 tests/index_out_of_bounds.out **114**

43 tests/invert.mpl **115**

44 tests/invert.out **116**

45 tests/join.mpl **117**

46 tests/join.out **118**

47 tests/print.mpl **119**

48 tests/print.out **120**

49 tests/print_void.mpl **121**

50 tests/print_void.out **122**

51 tests/range.mpl **123**

52 tests/range.out **124**

53 tests/str_concat.mpl **125**

54 tests/str_concat.out **126**

55 tests/str_conversion.mpl **127**

56 tests/str_conversion.out **128**

57 tests/str_tuple.mpl **129**

58 tests/str_tuple.out **130**

59 tests/test.txt **131**

60 tests/test1.mpl **132**

61 tests/test1.out **133**

62 tests/test2.mpl **134**

63 tests/test2.out **135**

64 tests/test3.mpl **136**

65 tests/test3.out **137**

66 tests/text_input.txt **138**

67 tests/undefined_function.mpl **139**

68 tests/undefined_function.out **140**

69 tests/undefined_variables.mpl **141**

70 tests/undefined_variables.out **142**

1 DaMPL_stdlib.mpl

```
fun readCSV(filename,tab){
        file = readfile(filename);
        table = strsplit(file,"\r\n");

        size = len(table);

        i=0;
        while(i < size){
                tab[] = strsplit(table[i],",");
                i = i+1;
        }

}


fun join(arr, separator){
        join__i = 1;

        join__size = len(arr);

        new_str = arr[0];

        while(join__i < join__size){
                new_str = new_str + separator + arr[join__i];
                join__i = join__i + 1;
        }

        return new_str;
}

fun join_tuple(tup, separator){
        join__tuple_i = 1;

        join__tuple_size = len(tup);

        new_str = tup$(0);

        while(join__tuple_i < join__tuple_size){
                new_str = new_str + separator + tup$(join__tuple_i);
                join__tuple_i = join__tuple_i + 1;
        }

        return new_str;
}

fun writeCSV(table,file){
        str = "";

        i = 0;
        size = len(table);
```

```
        while(i < size){
                str = str + join_tuple(table[i],",") + "\n";
                i = i+1;
        }

        writefile(file,str);
}

fun invert(arr){
        new_arr = [];

        i = len(arr) - 1;

        while(i >= 0){
                new_arr[] = arr[i];
                i = i-1;
        }

        return new_arr;
}

fun tuple_to_array(tup){
        arr = [];

        i = 0;

        while(i < len(tup)){
                arr[] = str(tup$(i));
                i = i+1;
        }

        return arr;
}

fun range(begin, end){
        i = begin;
        arr = [];

        while(i < end){
                arr[] = i;
                i = i + 1;
        }

        return arr;
}

fun range_s(begin, end, step){
        i = begin;
        arr = [];

        while(i < end){
                arr[] = i;
```

```
            i = i + step;
        }

        return arr;
}
```

2 helloworld.mpl

```
fun foo(a) {
        b=1;
        print("hi\n");
        lastone();
        return a;
}

fun bar(a) {
        c=1.1;
        print("Hello␣World!\n");
        /* foo(a); */ /* PROBLEMA QUANDO VARIAVEL LOCAL OU global    PASSADA
            COMO PARAMETRO */
        return foo(a);
}


fun lastone() {
        print("bye!\n");
}
```

3 libs/damplarray.c

```
#include "damplarray.h"
#include <stdio.h>

#define INITIAL_CAPACITY 4096

/* Constructor */

Array dampl_arr_new()
{
    Array arr = (Array) malloc (sizeof(tp_array_struct));
    arr->a = NULL;
    arr->size = 0;
    arr->capacity = 0;

    return arr;
}

/* Length */

int dampl_arr_len(Array arr)
{
    return arr->size;
}

/* Append */

void dampl_arr_append__arr (Array this, Array value)
{
    dampl_arr_ensure_cap__arr (this, this->size + 1);

    ((Array *) this->a)[this->size++] = value;
}

int dampl_arr_append__int (Array this, int value)
{
    dampl_arr_ensure_cap__int (this, this->size + 1);

    return ((int *) this->a)[this->size++] = value;
}

float dampl_arr_append__float (Array this, float value)
{
    dampl_arr_ensure_cap__float (this, this->size + 1);

    return ((float *) this->a)[this->size++] = value;
}

String dampl_arr_append__str (Array this, String value)
{
    dampl_arr_ensure_cap__str (this, this->size + 1);
```

```
    return ((String *) this->a)[this->size++] = value;
}

Tuple dampl_arr_append__tup (Array this, Tuple value)
{
    dampl_arr_ensure_cap__tup (this, this->size + 1);

    return ((Tuple *) this->a)[this->size++] = value;
}

/* Insert */

Array dampl_arr_insert__arr (Array this, int index, Array value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    dampl_arr_ensure_cap__arr (this, this->size + 1);

    /* Insert into array */

    int i;
    for (i = this->size++; i > index; i--)
    {
        ((Array *) this->a)[i] = ((Array *) this->a)[i - 1];
    }
    return ((Array *) this->a)[i] = value;
}

int dampl_arr_insert__int (Array this, int index, int value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    dampl_arr_ensure_cap__int (this, this->size + 1);

    /* Insert into array */

    int i;
    for (i = this->size++; i > index; i--)
    {
        ((int *) this->a)[i] = ((int *) this->a)[i - 1];
```

```
    }
    return ((int *) this->a)[i] = value;

}

float dampl_arr_insert__float (Array this, int index, float value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    dampl_arr_ensure_cap__float (this, this->size + 1);

    /* Insert into array */

    int i;
    for (i = this->size++; i > index; i--)
    {
        ((float *) this->a)[i] = ((float *) this->a)[i - 1];
    }
    return ((float *) this->a)[i] = value;
}

String dampl_arr_insert__str (Array this, int index, String value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    dampl_arr_ensure_cap__str (this, this->size + 1);

    /* Insert into array */

    int i;
    for (i = this->size++; i > index; i--)
    {
        ((String *) this->a)[i] = ((String *) this->a)[i - 1];
    }
    return ((String *) this->a)[i] = value;
}

Tuple dampl_arr_insert__tup (Array this, int index, Tuple value)
{
    /* Check border conditions */
```

```
    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    dampl_arr_ensure_cap__tup (this, this->size + 1);

    /* Insert into array */

    int i;
    for (i = this->size++; i > index; i--)
    {
        ((Tuple *) this->a)[i] = ((Tuple *) this->a)[i - 1];
    }
    return ((Tuple *) this->a)[i] = value;
}

/* Set */

Array dampl_arr_set__arr (Array this, int index, Array value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((Array *) this->a)[index] = value;
}

int dampl_arr_set__int (Array this, int index, int value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((int *) this->a)[index] = value;
}

float dampl_arr_set__float (Array this, int index, float value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
```

```
        exit(1);
    }

    return ((float *) this->a)[index] = value;
}

String dampl_arr_set__str (Array this, int index, String value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((String *) this->a)[index] = value;
}

Tuple dampl_arr_set__tup (Array this, int index, Tuple value)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((Tuple *) this->a)[index] = value;
}

/* Get */

Array dampl_arr_get__arr (Array this, int index)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((Array *) this->a)[index];
}

int dampl_arr_get__int (Array this, int index)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
```

```
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((int *) this->a)[index];
}

float dampl_arr_get__float (Array this, int index)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((float *) this->a)[index];
}

String dampl_arr_get__str (Array this, int index)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((String *) this->a)[index];
}

Tuple dampl_arr_get__tup (Array this, int index)
{
    /* Check border conditions */

    if (index >= this->size || index < 0)
    {
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }

    return ((Tuple *) this->a)[index];
}


/* Extract attribute from table (array of tuples) */

Array dampl_arr_extract_attr__int(Array table, int column){
    int i, data;
```

```
    Array arr = (Array) malloc (sizeof(tp_array_struct));
    arr->a = malloc(table->size*(sizeof(int)));
    arr->size = 0;
    arr->capacity = table->size;

    for(i = 0; i < table->size; i++){
        data = dampl_tup_get__int(dampl_arr_get__tup(table,i), column);
        dampl_arr_append__int(arr, data);
    }

    return arr;
}


Array dampl_arr_extract_attr__float(Array table, int column){
    int i;
    float data;

    Array arr = (Array) malloc (sizeof(tp_array_struct));
    arr->a = malloc(table->size*(sizeof(float)));
    arr->size = 0;
    arr->capacity = table->size;

    for(i = 0; i < table->size; i++){
        data = dampl_tup_get__float(dampl_arr_get__tup(table,i), column);
        dampl_arr_append__float(arr, data);
    }

    return arr;
}

Array dampl_arr_extract_attr__str(Array table, int column){
    int i;
    String data;

    Array arr = (Array) malloc (sizeof(tp_array_struct));
    arr->a = malloc(table->size*(sizeof(String)));
    arr->size = 0;
    arr->capacity = table->size;

    for(i = 0; i < table->size; i++){
        data = dampl_tup_get__str(dampl_arr_get__tup(table,i), column);
        dampl_arr_append__str(arr, data);
    }

    return arr;
}


/* Set attribute of table (array of tuples)
    Checks if array containing values are of
            same size as table */
```

```
Array dampl_arr_set_attr__int(Array table, int column, Array arr){
    int i;

    if(table->size != arr->size){
        fprintf(stderr, "Arrays with different sizes\n");
        exit(1);
    }

    for(i = 0; i < table->size; i++){
        dampl_tup_set__int(dampl_arr_get__tup(table,i), column,
            dampl_arr_get__int(arr,i));
    }

    return table;
}


Array dampl_arr_set_attr__float(Array table, int column, Array arr){
    int i;

    if(table->size != arr->size){
        fprintf(stderr, "Arrays with different sizes\n");
        exit(1);
    }

    for(i = 0; i < table->size; i++){
        dampl_tup_set__float(dampl_arr_get__tup(table,i), column,
            dampl_arr_get__float(arr,i));
    }

    return table;
}


Array dampl_arr_set_attr__str(Array table, int column, Array arr){
    int i;

    if(table->size != arr->size){
        fprintf(stderr, "Arrays with different sizes\n");
        exit(1);
    }

    for(i = 0; i < table->size; i++){
        dampl_tup_set__str(dampl_arr_get__tup(table,i), column,
            dampl_arr_get__str(arr,i));
    }

    return table;
}


/* Ensure capacity */
```

```
void dampl_arr_ensure_cap__arr (Array this, int sz)
{
    if (this->capacity == 0) {
        this->capacity = INITIAL_CAPACITY;
        Array *arr = (Array *) malloc (this->capacity * sizeof(Array));
        this->a = arr;
    }
    else if (this->capacity < sz)
    {
        this->capacity = sz * 10;

        /* Create new array */

        Array *arr = (Array *) malloc (this->capacity * sizeof(Array));

        /* Copy all elements to new array */

        int i;

        for (i = 0; i < this->size; i++)
        {
            arr[i] = ((Array *) this->a)[i];
        }

        free(this->a);

        this->a = arr;
    }
}

void dampl_arr_ensure_cap__int (Array this, int sz)
{
    if (this->capacity == 0) {
        this->capacity = INITIAL_CAPACITY;
        int *arr = (int *) malloc (this->capacity * sizeof(int));
        this->a = arr;
    }
    else if (this->capacity < sz)
    {
        this->capacity = sz * 10;

        /* Create new array */

        int *arr = (int *) malloc (this->capacity * sizeof(int));

        /* Copy all elements to new array */

        int i;

        for (i = 0; i < this->size; i++)
        {
            arr[i] = ((int *) this->a)[i];
        }
```

```
        free(this->a);

        this->a = arr;
    }
}

void dampl_arr_ensure_cap__float (Array this, int sz)
{
    if (this->capacity == 0) {
        this->capacity = INITIAL_CAPACITY;
        float *arr = (float *) malloc (this->capacity * sizeof(float));
        this->a = arr;
    }
    else if (this->capacity < sz)
    {
        this->capacity = sz * 10;

        /* Create new array */

        float *arr = (float *) malloc (this->capacity * sizeof(float));

        /* Copy all elements to new array */

        int i;

        for (i = 0; i < this->size; i++)
        {
            arr[i] = ((float *) this->a)[i];
        }

        free(this->a);

        this->a = arr;
    }
}

void dampl_arr_ensure_cap__str (Array this, int sz)
{
    if (this->capacity == 0) {
        this->capacity = INITIAL_CAPACITY;
        String *arr = (String *) malloc (this->capacity * sizeof(String));
        this->a = arr;
    }
    else if (this->capacity < sz)
    {
        this->capacity = sz * 10;

        /* Create new array */

        String *arr = (String *) malloc (this->capacity * sizeof(String));

        /* Copy all elements to new array */
```

```
        int i;

        for (i = 0; i < this->size; i++)
        {
            arr[i] = ((String *) this->a)[i];
        }

        free(this->a);

        this->a = arr;
    }
}

void dampl_arr_ensure_cap__tup (Array this, int sz)
{
    if (this->capacity == 0) {
        this->capacity = INITIAL_CAPACITY;
        Tuple *arr = (Tuple *) malloc (this->capacity * sizeof(Tuple));
        this->a = arr;
    }
    else if (this->capacity < sz)
    {
        this->capacity = sz * 10;

        /* Create new array */

        Tuple *arr = (Tuple *) malloc (this->capacity * sizeof(Tuple));

        /* Copy all elements to new array */

        int i;

        for (i = 0; i < this->size; i++)
        {
            arr[i] = ((Tuple *) this->a)[i];
        }

        free(this->a);

        this->a = arr;
    }
}


/* Concatenates arrays */

Array dampl_arr_concat__arr (Array arr1, Array arr2){
    int i;

    Array arr = dampl_arr_new();

    for(i = 0; i < arr1->size; i++){
```

```
        dampl_arr_append__arr (arr, dampl_arr_get__arr (arr1, i));
    }

    for(i = 0; i < arr2->size; i++){
        dampl_arr_append__arr (arr, dampl_arr_get__arr (arr2, i));
    }

    return arr;
}

Array dampl_arr_concat__int (Array arr1, Array arr2){
    int i;

    Array arr = dampl_arr_new();

    for(i = 0; i < arr1->size; i++){
        dampl_arr_append__int (arr, dampl_arr_get__int (arr1, i));
    }

    for(i = 0; i < arr2->size; i++){
        dampl_arr_append__int (arr, dampl_arr_get__int (arr2, i));
    }


    return arr;
}


Array dampl_arr_concat__float (Array arr1, Array arr2){
    int i;

    Array arr = dampl_arr_new();

    for(i = 0; i < arr1->size; i++){
        dampl_arr_append__float (arr, dampl_arr_get__float (arr1, i));
    }

    for(i = 0; i < arr2->size; i++){
        dampl_arr_append__float (arr, dampl_arr_get__float (arr2, i));
    }


    return arr;
}

Array dampl_arr_concat__str (Array arr1, Array arr2){
    int i;

    Array arr = dampl_arr_new();

    for(i = 0; i < arr1->size; i++){
        dampl_arr_append__str (arr, dampl_arr_get__str (arr1, i));
    }
```

```
    for(i = 0; i < arr2->size; i++){
        dampl_arr_append__str (arr, dampl_arr_get__str (arr2, i));
    }

    return arr;
}

Array dampl_arr_concat__tup (Array arr1, Array arr2){
    int i;

    Array arr = dampl_arr_new();

    for(i = 0; i < arr1->size; i++){
        dampl_arr_append__tup (arr, dampl_arr_get__tup (arr1, i));
    }

    for(i = 0; i < arr2->size; i++){
        dampl_arr_append__tup (arr, dampl_arr_get__tup (arr2, i));
    }

    return arr;
}


/* Set Range Functions */

int get_begin(int index1,int size){
    int begin;

    if(index1 == INT_MIN){
        begin = 0;
    }
    else if(index1 >= size){
        begin = size;
    }
    else if (index1 < 0){
        fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
        exit(1);
    }
    else{
        begin = index1;
    }

    return begin;
}

int get_end(int index2, int size){
    int end;

    if((index2 >= size) || (index2 == INT_MIN)){
        end = size;
```

```
    }
    else if (index2 < 0){
        fprintf(stderr, "Array out of bounds exception\n");
        exit(1);
    }
    else{
        end = index2;
    }

    return end;
}

Array dampl_arr_set_range__arr (Array this, int index1, int index2, Array
    other){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /**************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(Array)));
    /**************************************/

    for(i = 0; i < begin; i++){
        dampl_arr_append__arr (arr, dampl_arr_get__arr (this, i));
    }

    for(i = 0; i < other->size; i++){
        dampl_arr_append__arr (arr, dampl_arr_get__arr (other, i));
    }

    for(i = end; i < this->size; i++){
        dampl_arr_append__arr (arr, dampl_arr_get__arr (this, i));
    }

    this->a = arr->a;
    this->size = arr->size;
    this->capacity = arr->capacity;

    return this;
}
```

```
Array dampl_arr_set_range__int (Array this, int index1, int index2, Array
    other){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);

    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /***************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(int)));
    /***************************************/

    for(i = 0; i < begin; i++){
        dampl_arr_append__int (arr, dampl_arr_get__int (this, i));
    }

    for(i = 0; i < other->size; i++){
        dampl_arr_append__int (arr, dampl_arr_get__int (other, i));
    }

    for(i = end; i < this->size; i++){
        dampl_arr_append__int (arr, dampl_arr_get__int (this, i));
    }

    this->a = arr->a;
    this->size = arr->size;
    this->capacity = arr->capacity;

    return this;
}

Array dampl_arr_set_range__float (Array this, int index1, int index2, Array
    other){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }
```

```
    /*************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(Float)));
    /*************************************/

    for(i = 0; i < begin; i++){
        dampl_arr_append__float (arr, dampl_arr_get__float (this, i));
    }

    for(i = 0; i < other->size; i++){
        dampl_arr_append__float (arr, dampl_arr_get__float (other, i));
    }

    for(i = end; i < this->size; i++){
        dampl_arr_append__float (arr, dampl_arr_get__float (this, i));
    }


    this->a = arr->a;
    this->size = arr->size;
    this->capacity = arr->capacity;

    return this;
}

Array dampl_arr_set_range__str (Array this, int index1, int index2, Array
    other){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /*************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(String)));
    /*************************************/
```

```
    for(i = 0; i < begin; i++){
        dampl_arr_append__str (arr, dampl_arr_get__str (this, i));
    }

    for(i = 0; i < other->size; i++){
        dampl_arr_append__str (arr, dampl_arr_get__str (other, i));
    }

    for(i = end; i < this->size; i++){
        dampl_arr_append__str (arr, dampl_arr_get__str (this, i));
    }


    this->a = arr->a;
    this->size = arr->size;
    this->capacity = arr->capacity;

    return this;
}

Array dampl_arr_set_range__tup (Array this, int index1, int index2, Array
    other){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /***************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(Tuple)));
    /***************************************/

    for(i = 0; i < begin; i++){
        dampl_arr_append__tup (arr, dampl_arr_get__tup (this, i));
    }

    for(i = 0; i < other->size; i++){
        dampl_arr_append__tup (arr, dampl_arr_get__tup (other, i));
    }

    for(i = end; i < this->size; i++){
        dampl_arr_append__tup (arr, dampl_arr_get__tup (this, i));
    }
```

```
    this->a = arr->a;
    this->size = arr->size;
    this->capacity = arr->capacity;

    return this;
}


/* Get Range Functions */

Array dampl_arr_get_range__arr (Array this, int index1, int index2){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /**************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(Array)));
    /**************************************/

    for(i = begin; i < end; i++){
        dampl_arr_append__arr (arr, dampl_arr_get__arr (this, i));
    }

    return arr;
}

Array dampl_arr_get_range__int (Array this, int index1, int index2){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /**************************************/
    Array arr = dampl_arr_new();
```

```
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(int)));
    /************************************/

    for(i = begin; i < end; i++){
        dampl_arr_append__int (arr, dampl_arr_get__int (this, i));
    }

    return arr;
}

Array dampl_arr_get_range__float (Array this, int index1, int index2){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(Float)));
    /************************************/

    for(i = begin; i < end; i++){
        dampl_arr_append__float (arr, dampl_arr_get__float (this, i));
    }

    return arr;
}

Array dampl_arr_get_range__str (Array this, int index1, int index2){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }
```

```
    /************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(String)));
    /************************************/

    for(i = begin; i < end; i++){
        dampl_arr_append__str (arr, dampl_arr_get__str (this, i));
    }

    return arr;
}

Array dampl_arr_get_range__tup (Array this, int index1, int index2){
    int i;
    int begin, end;

    begin = get_begin(index1, this->size);
    end = get_end(index2, this->size);

    if(end < begin){
        end = begin;
    }

    /************************************/
    Array arr = dampl_arr_new();
    /* or */
    //int new_size = this->size - (end - begin) + other->size;
    //Array arr = (Array) malloc (sizeof(Array*));
    //arr->size = new_size;
    //arr->capacity = new_size;
    //arr->a = malloc(new_size*(sizeof(Tuple)));
    /************************************/

    for(i = begin; i < end; i++){
        dampl_arr_append__tup (arr, dampl_arr_get__tup (this, i));
    }

    return arr;
}


Tuple dampl_tup_convert(Array arr, int size, type_map* type){
    int i;

    if(arr->size != size){
        fprintf(stderr, "Incompatible␣sizes␣for␣array␣and␣tuple\n");
        exit(1);
```

```
    }

    Tuple tup = dampl_tup_new (size, type);

    for(i = 0; i < size; i++){
        dampl_tup_set__str(tup, i, dampl_arr_get__str (arr, i));
    }

    return tup;
}
```

4 libs/damplarray.h

```
#ifndef _DAMPL_ARRAY_
#define _DAMPL_ARRAY_

#include "damplstr.h"
#include "dampltup.h"
#include <stdlib.h>
#include <limits.h>

/* Array structure */

typedef struct
{
    void * a;
    int size;
    int capacity;
} tp_array_struct;

typedef tp_array_struct * Array;

/* Array constructor for every type */

Array dampl_arr_new();

/* Length function */

int dampl_arr_len(Array);

/* Array add method
   Appends an element in the end of the array */

void dampl_arr_append__arr (Array, Array);

int dampl_arr_append__int (Array, int);

float dampl_arr_append__float (Array, float);

String dampl_arr_append__str (Array, String);

Tuple dampl_arr_append__tup (Array, Tuple);


/* Array insertion method
   Insert an element in a given position, shifting all elements after it */

Array dampl_arr_insert__arr (Array, int, Array);

int dampl_arr_insert__int (Array, int, int);

float dampl_arr_insert__float (Array, int, float);
```

```
String dampl_arr_insert__str (Array, int, String);

Tuple dampl_arr_insert__tup (Array, int, Tuple);

/* Array set method
   Set a given position in an array to the value passed by the user */

Array dampl_arr_set__arr (Array, int, Array);

int dampl_arr_set__int (Array, int, int);

float dampl_arr_set__float (Array, int, float);

String dampl_arr_set__str (Array, int, String);

Tuple dampl_arr_set__tup (Array, int, Tuple);

/* Array get method
   Returns an element in a given position */

Array dampl_arr_get__arr (Array, int);

int dampl_arr_get__int (Array, int);

float dampl_arr_get__float (Array, int);

String dampl_arr_get__str (Array, int);

Tuple dampl_arr_get__tup (Array, int);


/* Extract attribute from table (array of tuples) */

Array dampl_arr_extract_attr__int(Array, int);

Array dampl_arr_extract_attr__float(Array, int);

Array dampl_arr_extract_attr__str(Array, int);

/* Set attribute of table (array of tuples)
        Checks if array containing values are of
                        same size as table */

Array dampl_arr_set_attr__int(Array, int, Array);

Array dampl_arr_set_attr__float(Array, int, Array);

Array dampl_arr_set_attr__str(Array, int, Array);

/* Array ensure capacity method */

void dampl_arr_ensure_cap__arr (Array, int);
```

```
void dampl_arr_ensure_cap__int (Array, int);
void dampl_arr_ensure_cap__float (Array, int);
void dampl_arr_ensure_cap__str (Array, int);
void dampl_arr_ensure_cap__tup (Array, int);

/* Concatenates arrays */
Array dampl_arr_concat__arr (Array, Array);
Array dampl_arr_concat__int (Array, Array);
Array dampl_arr_concat__float (Array, Array);
Array dampl_arr_concat__str (Array, Array);
Array dampl_arr_concat__tup (Array, Array);

/* Set Range Functions */
Array dampl_arr_set_range__arr (Array, int, int, Array);
Array dampl_arr_set_range__int (Array, int, int, Array);
Array dampl_arr_set_range__float (Array, int, int, Array);
Array dampl_arr_set_range__str (Array, int, int, Array);
Array dampl_arr_set_range__tup (Array, int, int, Array);

/* Get Range Functions */
Array dampl_arr_get_range__arr (Array, int, int);
Array dampl_arr_get_range__int (Array, int, int);
Array dampl_arr_get_range__float (Array, int, int);
Array dampl_arr_get_range__str (Array, int, int);
Array dampl_arr_get_range__tup (Array, int, int);

Tuple dampl_tup_convert(Array, int, type_map*);

#endif
```

5 libs/damplio.c

```
#include "damplio.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


Array files;


void dampl_print__str (String string)
{
    printf("%s", string);
}

void dampl_print__float (float f)
{
    printf("%g", f);
}

void dampl_print__int (int i)
{
    printf("%d", i);
}

void dampl_print__bool (int b){
  if (b) printf("true");
  else printf("false");
}

void dampl_print_arr__int (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);

    printf("[␣");

    if(dimensions == 1){
      dampl_print__int (dampl_arr_get__int (arr, 0));
      for(i = 1; i < size; i++){
        printf(",␣");
        dampl_print__int (dampl_arr_get__int (arr, i));
      }
    }
    else{
      for(i = 0; i < size; i++){
        dampl_print_arr__int (dampl_arr_get__arr (arr, i), dimensions-1);
      }
    }

    printf("␣]\n");
```

```
}

void dampl_print_arr__float (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);

    printf("[␣");

    if(dimensions == 1){
      dampl_print__float (dampl_arr_get__float (arr, 0));
      for(i = 1; i < size; i++){
        printf(",␣");
        dampl_print__float (dampl_arr_get__float (arr, i));
      }
    }
    else{
      for(i = 0; i < size; i++){
        dampl_print_arr__float (dampl_arr_get__arr (arr, i), dimensions-1);
      }
    }

    printf("␣]\n");
}

void dampl_print_arr__str (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);

    printf("[␣");

    if(dimensions == 1){
      dampl_print__str (dampl_arr_get__str(arr, 0));
      for(i = 1; i < size; i++){
        printf(",␣");
        dampl_print__str (dampl_arr_get__str(arr, i));
      }
    }
    else{
      for(i = 0; i < size; i++){
        dampl_print_arr__str (dampl_arr_get__arr(arr, i), dimensions-1);
      }
    }

    printf("␣]\n");
}

void dampl_print_arr__tup (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);
```

```
    printf("[␣");

    if(dimensions == 1){
      dampl_print__tup (dampl_arr_get__tup(arr, 0));
      for(i = 1; i < size; i++){
        printf(",␣");
        dampl_print__tup (dampl_arr_get__tup(arr, i));
      }
    }
    else{
      for(i = 0; i < size; i++){
        dampl_print_arr__tup (dampl_arr_get__arr (arr, i), dimensions-1);
      }
    }

    printf("␣]\n");
}


void dampl_print__tup (Tuple tup){
        int i, size;

        size = dampl_tup_len(tup);

  printf("(");
  dampl_print__str (dampl_tup_get__str (tup, 0));

  for(i = 1; i < size; i++){
    printf(",␣");
    dampl_print__str (dampl_tup_get__str (tup, i));
  }

  printf(")");
}


Array dampl_strsplit__str_str (String str, String separator){
        Array arr = dampl_arr_new();

        char *token;

  String new_str = dampl_str_copy(str);

        /* get the first token */
        token = strtok(new_str, separator);
  String new_token = dampl_str_copy(token);
        dampl_arr_append__str (arr, new_token);

        /* walk through other tokens */
        while((token = strtok(NULL, separator))) {
    String new_token = dampl_str_copy(token);
                dampl_arr_append__str (arr, new_token);
        }
```

```
        return arr;
}


// File Functions

String dampl_readfile__str (String file_name){
        FILE *fp;
        int i, count;
        char c;

        /* opening file for reading */
        fp = fopen(file_name, "r");
        if(fp == NULL) {
    perror("Error␣opening␣file");
    exit(-1);
        }

        /* Counts the number of characters on file */
        count = 0;
        while(fgetc(fp) != EOF){
                count++;
        }

        String file_str = (String) malloc((count+1)*sizeof(char));

        /* Goes to begining of the file */
        int err = fseek(fp, 0, SEEK_SET );
        if(err != 0) {
                perror("Error␣reading␣file");
        exit(-1);
  }

        for(i = 0; i < count; i++){
                c = fgetc(fp);
                file_str[i] = c;
        }

        file_str[count] = '\0';

        fclose(fp);

        return file_str;
}

void dampl_writefile__str_str (String file_name, String str){
        FILE *fp;


        /* opening file for reading */
        fp = fopen(file_name, "w");
        if(fp == NULL) {
```

```
        perror("Error␣opening␣file");
        exit(-1);
        }

        if(fputs(str, fp) == EOF){
                perror("Error␣writing␣file");
    exit(1);
  }

        fclose(fp);
}

void dampl_file_constructor(){

 files = dampl_arr_new();
}

void arr_ensure_cap__file (int sz){
    if (files->capacity < sz)
    {
        files->capacity = sz * 2;

        /* Create new array */

        FILE **arr = (FILE **) malloc (files->capacity * sizeof(FILE*));
        /* Copy all elements to new array */

        int i;

        for (i = 0; i < files->size; i++){
            arr[i] = (FILE*)((FILE **) files->a)[i];
        }

        files->a = arr;
    }
}

void arr_append__file (FILE* value){

    arr_ensure_cap__file (files->size + 1);

    ((FILE**)files->a)[files->size++] = value;
}

FILE *arr_get__file (int index){
    /* Check border conditions */

    // if (index >= this->size || index < 0){
    //     fprintf(stderr, "Array␣out␣of␣bounds␣exception\n");
    //     exit(1);
    // }

    return  ((FILE **) files->a)[index];
```

```
}

int dampl_open__str_str(String filename,String mode){

  FILE **fp = malloc(sizeof(FILE*));

  *fp = fopen(filename, mode);
  if(fp == NULL) {
    perror("Error␣opening␣file");
    exit(-1);
  }

  arr_append__file (*fp);

  return files->size-1;
}

void dampl_close__int(int index){

  FILE* fp;

  if (index >= files->size || index < 0){
    fprintf(stderr, "Illegal file descriptor\n");
    exit(1);
  }

  fp = (FILE*) arr_get__file (index);

  fclose(fp);
}

String dampl_readline__int(int index){
  char c;
  int i;
  FILE* fp;

  String file_str;

  int eof_flag = 1;
  int count = 0;

  if (index >= files->size || index < 0){
    fprintf(stderr, "Illegal␣file␣descriptor\n");
    exit(1);
  }

  fp = (FILE*) arr_get__file (index);

  while((c = fgetc(fp)) != EOF){
    if(c == '\n') break;
    count++;
  }
```

```
  if(c != '\n') eof_flag = 0;

  file_str = (String) malloc((count+1)*sizeof(char));


  int err = fseek(fp, -(count+eof_flag), SEEK_CUR );
  if(err != 0) {
    perror("Error reading file");
    exit(-1);
  }

  for(i = 0; i < count; i++){
    c = fgetc(fp);
    file_str[i] = c;
  }

  c = fgetc(fp);
  file_str[count] = '\0';

  return file_str;
}

void dampl_writeline__int_String(int index, String str){

  FILE* fp;

  if (index >= files->size || index < 0){
    fprintf(stderr, "Illegal file descriptor\n");
    exit(1);
  }


  fp = (FILE*) arr_get__file (index);

  if(fputs(str, fp) == EOF){
    perror("Error writing file");
    exit(1);
  }
}
```

6 libs/damplio.h

```
#ifndef _DAMPL_IO_
#define _DAMPL_IO_

#include "damplstr.h"
#include "damplarray.h"

/*
 * Receives a string as parameter to be printed.
 */
void dampl_print__str (String string);

void dampl_print__float (float f);

void dampl_print__int (int i);

void dampl_print__bool (int);

/* Prints array */

void dampl_print_arr__int (Array, int);

void dampl_print_arr__float (Array, int);

void dampl_print_arr__str (Array, int);

void dampl_print_arr__tup (Array, int);


/* Prints Tuple */
void dampl_print__tup (Tuple);


Array dampl_strsplit__str_str (String, String);

String dampl_readfile__str (String);

void dampl_writefile__str_str (String, String);


void dampl_file_constructor();

int dampl_open__str_str(String, String);

void dampl_close__int(int);

String dampl_readline__int(int);

void dampl_writeline__int_String(int, String);

#endif
```

7 libs/dampllib.c

```
#include "dampllib.h"
#include <stdio.h>
#include <stdlib.h>

String dampl_str__str (String string) {
        return string;
}

String dampl_str__float (float f){
        int size = snprintf(NULL, 0, "%g", f);
        char *s = malloc(size + 1);
        sprintf(s, "%g", f);
        return s;
}

String dampl_str__int (int i) {
        int size = snprintf(NULL, 0, "%d", i);
        char *s = malloc(size + 1);
        sprintf(s, "%d", i);
        return s;
}

String dampl_str__bool (int b){
  String str;
  if (b) str = dampl_str_copy("true");
  else str = dampl_str_copy("false");

  return str;
}


String dampl_str_arr__int (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);

    String str;

    str = dampl_str_copy("[␣");

    if(dimensions == 1){
      str = dampl_str_concat(str, dampl_str__int (dampl_arr_get__int (arr, 0)
          ));
      for(i = 1; i < size; i++){
        str = dampl_str_concat(str,",␣");
        str = dampl_str_concat(str,dampl_str__int (dampl_arr_get__int (arr, i
            )));
      }
    }
    else{
```

```
      for(i = 0; i < size; i++){
        str = dampl_str_concat(str,dampl_str_arr__int (dampl_arr_get__arr (
            arr, i), dimensions-1));
      }
    }

    str = dampl_str_concat(str,"␣]");

    return str;
}


String dampl_str_arr__float (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);

    String str;

    str = dampl_str_copy("[␣");

    if(dimensions == 1){
      str = dampl_str_concat(str, dampl_str__float (dampl_arr_get__float (arr
          , 0)));
      for(i = 1; i < size; i++){
        str = dampl_str_concat(str,",␣");
        str = dampl_str_concat(str,dampl_str__float (dampl_arr_get__float (
            arr, i)));
      }
    }
    else{
      for(i = 0; i < size; i++){
        str = dampl_str_concat(str,dampl_str_arr__float (dampl_arr_get__arr (
            arr, i), dimensions-1));
      }
    }

    str = dampl_str_concat(str,"␣]");

    return str;
}

String dampl_str_arr__str (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);

    String str;

    str = dampl_str_copy("[␣");

    if(dimensions == 1){
      str = dampl_str_concat(str, dampl_arr_get__str (arr, 0));
```

```
      for(i = 1; i < size; i++){
        str = dampl_str_concat(str,",␣");
        str = dampl_str_concat(str, dampl_arr_get__str (arr, i));
      }
    }
    else{
      for(i = 0; i < size; i++){
        str = dampl_str_concat(str,dampl_str_arr__str (dampl_arr_get__arr (
            arr, i), dimensions-1));
      }
    }

    str = dampl_str_concat(str,"␣]");

    return str;
}

String dampl_str_arr__tup (Array arr, int dimensions){
    int i, size;

    size = dampl_arr_len(arr);

    String str;

    str = dampl_str_copy("[␣");

    if(dimensions == 1){
      str = dampl_str_concat(str, dampl_arr_get__str (arr, 0));
      for(i = 1; i < size; i++){
        str = dampl_str_concat(str,",␣");
        str = dampl_str_concat(str, dampl_arr_get__str (arr, i));
      }
    }
    else{
      for(i = 0; i < size; i++){
        str = dampl_str_concat(str,dampl_str_arr__str (dampl_arr_get__arr (
            arr, i), dimensions-1));
      }
    }

    str = dampl_str_concat(str,"␣]");

    return str;
}

String dampl_str__tup (Tuple tup){
        int i, size;

        size = dampl_tup_len(tup);

        String str;

    str = dampl_str_copy("(");
```

```
        str = dampl_str_concat(str, dampl_tup_get__str (tup, 0));

        for(i = 1; i < size; i++){
                str = dampl_str_concat(str, ", ");
                str = dampl_str_concat(str, dampl_tup_get__str (tup, i));
        }

        str = dampl_str_concat(str,")");

        return str;
}

Array build_args_array(String* args) {
        Array a = dampl_arr_new();
        while(*args) {
                dampl_arr_append__str(a,*args);
                args++;
        }
        return a;
}


void dampl_die__(void) {
        exit(0);
}
```

8 libs/dampllib.h

```
#ifndef _DAMPL_LIB_
#define _DAMPL_LIB_

#include "damplio.h"
#include "damplstr.h"
#include "damplarray.h"
#include "dampltup.h"


String dampl_str__str (String string);

String dampl_str__float (float f);

String dampl_str__int (int i);

String dampl_str__bool (int);


String dampl_str_arr__int (Array, int);

String dampl_str_arr__float (Array, int);

String dampl_str_arr__str (Array, int);

String dampl_str_arr__tup (Array, int);

String dampl_str__tup (Tuple);

Array build_args_array(String*);


void dampl_die__(void);

#endif
```

9 libs/damplstr.c

```
#include <string.h>
#include <stdlib.h>
#include "damplstr.h"


String dampl_str_concat(String s1, String s2) {
        int size = strlen(s1) + strlen(s2);
        char *s = malloc(size+1);
        strcpy(s,s1);
        strcat(s,s2);
        return s;
}

String dampl_str_copy(String s1) {
        int size = strlen(s1);
        char *s = malloc(size+1);
        strcpy(s,s1);
        return s;
}

int dampl_len__str(String s1){
        return strlen(s1);
}
```

10 libs/damplstr.h

```
#ifndef _DAMPL_STR_
#define _DAMPL_STR_


typedef char* String;

String dampl_str_concat(String s1, String s2);

String dampl_str_copy(String s1);

int dampl_len__str(String s1);

#endif
```

11 libs/dampltup.c

```
#include <stdlib.h>
#include <stdio.h>
#include "dampltup.h"

Tuple dampl_tup_new (int size, type_map* type){
        int i;

        Tuple tup = malloc(sizeof(tp_tup_struct));
        tup->size = size;
    tup->map = type;


        tup->values = malloc(size*sizeof(String));

        for(i = 0; i < size; i++){
        switch(type[i]){
            case real:
                tup->values[i] = dampl_str_copy("0.0");
                break;
            case integer:
                tup->values[i] = dampl_str_copy("0");
                break;
            case text:
                tup->values[i] = dampl_str_copy("");
                break;
        }

        }

        return tup;
}


/* Length function */

int dampl_tup_len(Tuple tup){
        return tup->size;
}

/* Tuple insertion method */

int dampl_tup_set__int(Tuple tup, int index, int data){

        /* Check border conditions */

        if (index >= tup->size || index < 0)
    {
        fprintf(stderr, "Out␣of␣bounds␣exception\n");
        exit(1);
    }
```

```
        char str[11];

        sprintf(str, "%d", data);

        tup->values[index] = dampl_str_copy(str);

        return data;
}

float dampl_tup_set__float(Tuple tup, int index, float data){

        /* Check border conditions */

        if (index >= tup->size || index < 0)
    {
        fprintf(stderr, "Out␣of␣bounds␣exception\n");
        exit(1);
    }

        char str[64];

        sprintf(str, "%g", data);

        tup->values[index] = dampl_str_copy(str);

        return data;
}

String dampl_tup_set__str(Tuple tup, int index, String data){
    char *eptr;
    double result_d;
    long result_l;


    switch(tup->map[index]){
        case real:
            /* Convert the provided value to a double */
            result_d = strtod(data, &eptr);
            dampl_tup_set__float(tup, index, (float)result_d);
            break;

        case integer:
            /* Convert the provided value to a long */
            result_l = strtol(data, &eptr, 10);
            dampl_tup_set__int(tup, index, (int)result_l);
            break;

        case text:
            /* Check border conditions */

            if (index >= tup->size || index < 0){
                fprintf(stderr, "Out␣of␣bounds␣exception\n");
```

```
                exit(1);
            }

            tup->values[index] = dampl_str_copy(data);
            break;
    }

        return tup->values[index];
}


int dampl_tup_get__int(Tuple tup, int index){
        String str;
        char *eptr;
    long result;

        /* Check border conditions */

    if (index >= tup->size || index < 0)
    {
        fprintf(stderr, "Out␣of␣bounds␣exception\n");
        exit(1);
    }

    str = tup->values[index];

    /* Convert the provided value to a decimal long long */
    result = strtol(str, &eptr, 10);

    return (int)result;
}

float dampl_tup_get__float(Tuple tup, int index){
        char *str;
        char *eptr;
    double result;

        /* Check border conditions */

    if (index >= tup->size || index < 0)
    {
        fprintf(stderr, "Out␣of␣bounds␣exception\n");
        exit(1);
    }

    str = tup->values[index];

    /* Convert the provided value to a double*/
    result = strtod(str, &eptr);

    return (float)result;
}
```

```
String dampl_tup_get__str(Tuple tup, int index){

        /* Check border conditions */

        if (index >= tup->size || index < 0)
    {
        fprintf(stderr, "Out␣of␣bounds␣exception\n");
        exit(1);
    }

        return tup->values[index];
}
```

12 libs/dampltup.h

```
#ifndef _DAMPL_TUP_
#define _DAMPL_TUP_

/* Array structure */

#include "damplstr.h"
#include <stdlib.h>

typedef enum{
        real,
        integer,
        text
} type_map;

typedef struct
{
    String* values;
    int size;
    type_map* map;
} tp_tup_struct;

typedef tp_tup_struct * Tuple;


//Tuple dampl_tup_new (int size, type_map);
Tuple dampl_tup_new (int size, type_map* type);


/* Length function */

int dampl_tup_len(Tuple);


/* Tuple insertion method */

int dampl_tup_set__int(Tuple, int, int);

float dampl_tup_set__float(Tuple, int, float);

String dampl_tup_set__str(Tuple, int, String);


int dampl_tup_get__int(Tuple, int);

float dampl_tup_get__float(Tuple, int);

String dampl_tup_get__str(Tuple, int);


#endif
```

13 libs/Makefile

```
CC = gcc
CFLAGS = -Wall -O2
LDFLAGS =

libdamplio.a: damplio.o damplstr.o dampllib.o dampltup.o damplarray.o
        ar rc libdampllib.a damplio.o damplstr.o dampllib.o dampltup.o
            damplarray.o
        ranlib libdampllib.a

damplio.o: damplio.h
damplstr.o: damplstr.h
dampllib.o: dampllib.h
dampltup.o: dampltup.h
damplarray.o: damplarray.h

.PHONY: clean
clean:
        rm -f *.o *.a

.PHONY: all
all: clean libdampllib.a
```

14 Makefile

```
.PHONY: project
project:
	cd src; make; cp dampl ../dampl
	cd libs; make
	./dampl mycsv.mpl > mycsv.c
	gcc mycsv.c -O2 -o mycsv -Ilibs/ -Llibs/ -ldampllib

.PHONY: clean
clean:
	rm -f *.c hello dampl sandbox
	cd src; make clean
	cd libs; make clean
```

15 mycsv.mpl

```
include "DaMPL_stdlib.mpl";

tuple Report {cdatetime, address,district:integer,beat,grid:integer,
    crimedescr,ucr_ncic_code:integer, latitude:real,longitude:real}


tab = Report[];



readCSV("SacramentocrimeJanuary2006.csv", tab);



tab = tab[1:];


arr = tab$cdatetime;

size = len(arr);

i = 0;

writefile("out.txt",join(arr, "\n"));



while (i<size){
        arr[i] = str(i);

        i = i+1;
}

writefile("out2.txt",join(arr, "\n"));


tab$cdatetime = arr;

writeCSV(tab,"tst.csv");
```

16 sandbox.mpl

```
tuple Foo{a,b,c}

fun foo(a) {
        arr=["a","c","dd"];
        a[]=arr;
}


a=Foo[];
foo(a);
```

17 src/ast.ml

```
(* Abstract Syntax Tree *)


type typ = Undefined | Bool | Int | Float | Void | String | Tuple of string |
     Table of string | Array of typ


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

type uop = Neg | Not


type tupitem = typ * string

type tup = string * tupitem list

type obj = (* lhs *)
    Id of string
  | Brac of obj * expr * bool (* a[0] a[i] a[i+1] *)
  | Brac2 of obj * expr * expr (* a[0:2] *)
  | Attr of obj * string (* a$b *)
  | AttrInx of obj * expr (* a$(0) *)
and
 expr =
    Literal of int
  | BoolLit of bool
  | FloatLit of float
  | StrLit of string
  | Obj of obj
  | Binop of expr * op * expr(* as on lhs *)
  | Unop of uop * expr
  | Assign of obj * expr
  | Call of string * expr list
  | TupInst of string (* tuple instantiation *)
  | TabInst of string (* table instantiation e.g. Foo[] *)
  | Arr of expr list (* arrays e.g. [1,2,3] *)
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of string * expr * stmt (* for i in a *)
  | While of expr * stmt
  | Break
  | Continue

type func_decl = {
```

```
    fname : string;
    formals : string list;
    locals : string list;
    body : stmt list;
  }

type program_decl =
    Func of func_decl
  | Tup of tup

type program = stmt list * program_decl list

type includ =
  FileIncl of string

type program_with_headers = includ list * program
```

18 src/builtin.ml

```
open Ast
open Semt

let built_in_prototypes = [
  (Void,"print",[
    [String];
    [Int];
    [Float];
    [Bool];
  ]);
  (String,"str",[
    [String];
    [Int];
    [Float];
    [Bool];
  ]);
  (Float,"float",[
    [String];
    [Int];
    [Float];
  ]);
  (Int,"int",[
    [String];
    [Int];
    [Float];
  ]);
  (Int,"len",[
    [String];
  ]);
  (Void,"die",[[]]);
  (String,"readfile",[
    [String]
  ]);
  (Array(String),"strsplit",[
    [String;String]
  ]);
  (Void,"writefile",[
    [String;String];
  ]);
  (Int,"open",[
    [String;String];
  ]);
  (Void,"close",[
    [Int];
  ]);
  (String,"readline",[
    [Int];
  ]);
  (Void,"writeline",[
    [Int;String];
```

```
  ]);
];;

let get_built_in_rtyp = function
    (t,_,_) -> t

let get_built_in_name = function
    (_,name,_) -> name

let get_built_in_formals = function
    (_,_,formals) -> formals
```

19 src/codegen.ml

```
(* Code generation: translate takes a semantically checked Semantic ST and
produces C

*)

open Ast

open Semt

module StringMap = Map.Make(String)

let simple_string_of_typ = function
    Bool -> "int"
  | Int -> "int"
  | Float -> "float"
  | Void -> "void"
  | String -> "str"
  | Array(_) -> "arr"
  | Table(_) -> "arr"
  | Tuple(_) -> "tup"
  | _ -> raise(Failure("simple_string_of_typ failure"))

let string_of_typ = function
    Bool -> "int"
  | Int -> "int"
  | Float -> "float"
  | Void -> "void"
  | String -> "String"
  | Array(_) -> "Array"
  | Table(_) -> "Array"
  | Tuple(_) -> "Tuple"
  | Undefined -> raise(Failure("Undefined type on string_of_typ"))


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

let string_of_uop = function
    Neg -> "-"
```

```
  | Not -> "!"

let set_or_insert t =
  if t then "insert" else "set"

let empty_inx = SString("INT_MIN")

let mapping_of_tup (t : tup) : string =
  let string_of_attr_type t = ( match fst t with
      String -> "text"
    | Float -> "real"
    | Int -> "integer"
    | _ -> raise(Failure("tuple␣mapping␣failure")) ) in
  "type_map␣map"^fst t^"[]␣=␣{"
    ^(String.concat ",␣" (List.map string_of_attr_type (snd t)))^"};\n"

let rec string_of_obj = function
    SId(s) -> "dampl_" ^ s
  | SBrac(o,e,_,t) -> "dampl_arr_get__" ^ simple_string_of_typ t ^ "("
      ^ string_of_obj o ^ "," ^ string_of_expr e ^ ")"
  | SBrac2(o,e1,e2,typ) ->
      let e1 = if e1 = SNoexpr then empty_inx else e1 in
      let e2 = if e2 = SNoexpr then empty_inx else e2 in
      let arrtyp = (match typ with
              Array(t) -> t
            | Table(tname) -> Tuple(tname)
            | _ -> raise(Failure("attr␣of␣table␣failure"))
          ) in
        "dampl_arr_get_range__"^simple_string_of_typ arrtyp^"(" ^
            string_of_obj o ^ ","
      ^ string_of_expr e1 ^ "," ^ string_of_expr e2 ^ ")"
  | SAttr(otyp,atyp,o,name,inx) -> (match otyp with
        Tuple(tname) -> "dampl_tup_get__"^simple_string_of_typ atyp^"("
          ^string_of_obj o^","^string_of_int inx^")"
      | Table(tname) -> let typ = (match atyp with
            Array(t) -> simple_string_of_typ t
          | _ -> raise(Failure("attribute␣extraction␣failure"))
        ) in "dampl_arr_extract_attr__"^typ^"("^string_of_obj o^","
          ^string_of_int inx^")"
      | _ -> raise(Failure("$attribute␣failure"))
    )
  | SAttrInx(otyp,atyp,o,inx_expr) -> (match otyp with
        Tuple(tname) -> "dampl_tup_get__str("
          ^string_of_obj o^","^string_of_expr inx_expr^")"
      | Table(tname) -> "dampl_arr_extract_attr__str("^string_of_obj o^","
          ^string_of_expr inx_expr^")"
      | _ -> raise(Failure("$attribute␣failure"))
    )
  (*
  | SBrac of string * sem_expr (* a[0] a[i] a[i+1] *)
  | SBrac2 of string * sem_expr * sem_expr (* a[0:2] *)
  | SAttr of string * string (* a$b *)
  *)
```

```
and string_of_expr = function
    SLiteral(l) -> string_of_int l
  | SBoolLit(true) -> "1"
  | SBoolLit(false) -> "0"
  | SFloatLit(f) -> string_of_float f
  | SStrLit(s) -> s
  | SObj(o) -> string_of_obj o
  | SBinop(t, e1, o, e2) -> ( match t with
        Array(t2) -> "(␣dampl_arr_concat__"^simple_string_of_typ t2^"("^
            string_of_expr e1
          ^ "," ^ string_of_expr e2 ^ ")␣)"
      | String -> ( match o with
            Add -> "(␣dampl_str_concat("^ string_of_expr e1 ^ "," ^
                string_of_expr e2 ^ ")␣)"
          | Equal | Neq | Less | Leq | Greater | Geq ->
              "(␣strcmp("^ string_of_expr e1 ^ "," ^ string_of_expr e2 ^ ")␣"
                  ^string_of_op o^"␣0␣)"
          | _ -> raise(Failure("String␣operation␣translation␣failure"))
      )
      | _ -> "(" ^ string_of_expr e1 ^ "␣" ^ string_of_op o ^ "␣" ^
          string_of_expr e2 ^ ")"
    )
  | SUnop(t, o, e) -> "(" ^ string_of_uop o ^ string_of_expr e ^ ")"
  | SAssign(t, o, e) -> ( match o with
        SBrac(o2,e2,is_ins,_) -> ( if e2 = SNoexpr
          then ( "dampl_arr_append__" ^ simple_string_of_typ t
         ^ "(" ^ string_of_obj o2 ^ "," ^ string_of_expr e ^ ")")
         else ( "dampl_arr_" ^ set_or_insert is_ins ^ "__" ^
             simple_string_of_typ t
         ^ "(" ^ string_of_obj o2 ^ "," ^ string_of_expr e2 ^ "," ^
             string_of_expr e ^ ")" )
      )
      | SBrac2(o2,e21,e22,typ) ->
        let e21 = if e21 = SNoexpr then empty_inx else e21 in
        let e22 = if e22 = SNoexpr then empty_inx else e22 in
        let arrtyp = (match typ with
              Array(t) -> t
            | Table(tname) -> Tuple(tname)
            | _ -> raise(Failure("attr␣of␣table␣failure"))
          ) in
        "dampl_arr_set_range__"^simple_string_of_typ arrtyp^"(" ^
            string_of_obj o2 ^ ","
        ^ string_of_expr e21 ^ "," ^ string_of_expr e22 ^ "," ^
            string_of_expr e ^ ")"
      | SAttr(otyp,atyp,o2,name,inx) -> (match otyp with
          Tuple(tname) -> "dampl_tup_set__"^simple_string_of_typ t^"("
            ^string_of_obj o2^","^string_of_int inx^","^string_of_expr e^")"
        | Table(tname) -> let attrtyp = (match atyp with
              Array(t) -> t
            | _ -> raise(Failure("attr␣of␣table␣failure"))
          ) in
          "dampl_arr_set_attr__"^simple_string_of_typ attrtyp^"("
```

```
          ^string_of_obj o2^","^string_of_int inx^","^string_of_expr e^")"
        | _ -> raise(Failure("$attribute␣failure"))
      )
      | SAttrInx(otyp,atyp,o2,inx_expr) -> (match otyp with
          Tuple(tname) -> "dampl_tup_set__"^simple_string_of_typ t^"("
            ^string_of_obj o2^","^string_of_expr inx_expr^","^string_of_expr
                e^")"
        | Table(tname) -> "dampl_arr_set_attr__"^simple_string_of_typ t^"("
          ^string_of_obj o^","^string_of_expr inx_expr^","^string_of_expr e^"
              )"
        | _ -> raise(Failure("$attribute␣failure"))
      )
      | _ -> string_of_obj o ^ "␣=␣" ^ string_of_expr e
  )
  | SCall(f, el) ->
      "dampl_" ^ f ^ "(" ^ String.concat ",␣" (List.map string_of_expr el) ^
          ")"
  | SSpecialCall(_,f,el) ->
      f ^ "(" ^ String.concat ",␣" (List.map string_of_expr el) ^ ")"
  | SArr(t,exprs) -> if (List.length exprs) = 0
      then "dampl_arr_new()"
      else (
        let create_append expr =
            "dampl_arr_append__"^simple_string_of_typ t^"(a,"^string_of_expr
                expr^");\n"
        in "({\nArray␣a=dampl_arr_new();\n"
          ^(String.concat "" (List.map create_append exprs))
          ^"a;})"
      )
  | STabInst(_) -> "dampl_arr_new()"
  | STupInst(name,n) -> "dampl_tup_new("^string_of_int n^",map"^name^")"
  (*
  | STupInst of string (* tuple instantiation *)
  | STabInst of string (* table instantiation e.g. Foo[] *)
  | STupInit of string * expr list (* tuple init e.g. Foo{1,2,"abc"} *)
  | SArr of expr list (* arrays e.g. [1,2,3] *)
  | SDict of expr list * expr list (* dicts *)
  *)
  | SNoexpr -> ""
  | SString(s) -> s


and string_of_stmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_expr expr ^ ";\n";
  | SReturn(expr) -> "return␣" ^ string_of_expr expr ^ ";\n";
  | SIf(e, s, SBlock([])) -> "if␣(" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s
  | SIf(e, s1, s2) ->  "if␣(" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2

  | SFor(str,t, e, s) -> "{\n"
```

```
      ^ string_of_typ t ^ "␣dampl_"^str^"␣=␣dampl_arr_get__"^
        simple_string_of_typ t^"("^ string_of_expr e ^",0);\n"
      ^ "int␣i_"^str ^ "␣=␣0;\n"
      ^ "while(i_"^str^"␣<␣dampl_arr_len(" ^ string_of_expr e ^ ")␣)␣{\n"
        ^"dampl_"^str^"␣=␣dampl_arr_get__"^simple_string_of_typ t^"("^
          string_of_expr e ^",i_"^str ^ ");\n"
      ^string_of_stmts ( match s with SBlock(sl) -> sl | stmt -> [stmt] )
      ^"i_"^str^"++;\n}\n}\n"

     (* "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
    *)
  | SWhile(e, s) -> "while␣(" ^ string_of_expr e ^ ")␣" ^ string_of_stmt s
  | SBreak -> "break;\n"
  | SContinue -> "continue;\n"

and string_of_stmts stmts =
  String.concat "" (List.map string_of_stmt stmts)


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

let string_of_formal f = (string_of_typ (fst f))^"␣dampl_"^(snd f)

let string_of_global global =
  string_of_typ (fst global) ^ "␣" ^ "␣dampl_" ^ (snd global) ^";\n";;

let fdecl_prototype fdecl =
  string_of_typ fdecl.rtyp ^ "␣" ^ "␣dampl_" ^ fdecl.semfname ^
  "(" ^ String.concat ",␣" (List.map string_of_formal fdecl.semformals) ^
  ");\n";;

let string_of_fdecl fdecl =
  string_of_typ fdecl.rtyp ^ "␣" ^ "␣dampl_" ^ fdecl.semfname ^
  "(" ^ String.concat ",␣" (List.map string_of_formal fdecl.semformals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl (StringMap.bindings !(fdecl.
      semlocals) ) ) ^
  String.concat "" (List.map string_of_stmt fdecl.sembody) ^
  "}\n\n"



let string_of_program (globals, statements, functions, tuples) =
  "#include␣<stdio.h>\n" ^ "#include␣<stdlib.h>\n" ^
  "#include␣\"dampllib.h\"\n\n" ^
  String.concat "" (List.map mapping_of_tup tuples) ^ "\n" ^
  String.concat "" (List.map string_of_global globals) ^ "\n" ^
  String.concat "" (List.map fdecl_prototype functions) ^ "\n" ^
  String.concat "" (List.map string_of_fdecl functions) ^
  "int␣main(int␣argc,char**␣argv){\n" ^
  "dampl_args=build_args_array(argv);\n" ^
  "dampl_file_constructor();\n" ^
```

```
String.concat "" (List.map string_of_stmt statements) ^
"return␣0;\n}\n"
```

20 src/main.ml

```
open Ast
open Parser
open Semt

let already_included = ref []

let basepath = ref ""

let remove_quotes str =
  String.sub str 1 ((String.length str)-2)

let get_file_from_include incl = (!basepath)^(match incl with
  FileIncl(f) -> remove_quotes f
)

let rec join_ast_list l = match l with
    a::tl -> let rec_call = join_ast_list tl in
      ((fst a)@(fst rec_call),(snd a)@(snd rec_call))
  | [] -> ([],[])

let rec get_program_from_include incl =
  let filename = get_file_from_include incl in
  if not (List.mem filename !already_included) then (
    already_included := filename::(!already_included);
    let file = Pervasives.open_in filename in
    let lexbuf = Lexing.from_channel file in
    let includes,ast = Parser.program Scanner.token lexbuf in
    join_ast_list ((List.map get_program_from_include includes)@[ast])
  ) else ([],[])

let _ =
  let filename =
    if (Array.length Sys.argv) = 2 then Sys.argv.(1)
    else raise (Invalid_argument("Usage:␣./dampl␣<input_filename>")) in
  already_included := filename::(!already_included);
  basepath := if String.contains filename '/' then
    String.sub filename 0 ((String.rindex filename '/')+1) else "";
  let file = Pervasives.open_in filename in
  let lexbuf = Lexing.from_channel file in
  let includes,ast = Parser.program Scanner.token lexbuf in
  let complete_ast = join_ast_list ((List.map get_program_from_include
     includes)@[ast]) in
  let semt = Sconv.convert complete_ast in
  print_string (Codegen.string_of_program semt);;
```

21 src/Makefile

```
TARFILES = Makefile scanner.mll parser.mly ast.mli calc.ml

OBJS = parser.cmo scanner.cmo semant.cmo semt.cmo builtin.cmo codegen.cmo
    sconv.cmo main.cmo

dampl : $(OBJS)
        ocamlc -o dampl $(OBJS)

scanner.ml : scanner.mll
        ocamllex scanner.mll

parser.ml parser.mli : parser.mly
        ocamlyacc -v parser.mly

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

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

calculator.tar.gz : $(TARFILES)
        cd .. && tar zcf calculator/calculator.tar.gz $(TARFILES:%=calculator
            /%)

.PHONY : clean
clean :
        rm -f dampl main tokenizer parser.ml parser.mli scanner.ml *.cmo *.
            cmi

.PHONY : all
all : clean dampl

# Generated by ocamldep *.ml *.mli
dampl.cmo: scanner.cmo semant.cmo parser.cmi sconv.cmo codegen.cmo ast.cmo
    semt.cmo
dampl.cmx: scanner.cmx semant.cmx parser.cmx sconv.cmx codegen cmx ast.cmo
    semt.cmx
codegen.cmo: semt.cmo ast.cmo
codegen.cmx: semt.cmx ast.cmx
builtin.cmo: semt.cmo ast.cmo
builtin.cmx: semt.cmx ast.cmx
sconv.cmo: semt.cmo ast.cmo
sconv.cmx: semt.cmx ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmo parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```

22 src/parser.mly

```
/* Ocamlyacc parser for DaMPL */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA COLON AT
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE
%token INCLUDE TUPLE DOLLAR BREAK CONTINUE FUN IN
%token REAL INTEGER TEXT
%token <int> LITERAL
%token <float> FLOAT
%token <string> STRING
%token <string> ID
%token <string> TID
%token EOF

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


%start program
%type <Ast.program_with_headers> program

%%

program:
  program_with_headers EOF { $1 }

program_with_headers:
    includs decls { $1, $2 }
  | decls { [], ($1) }

includs:
    includs includ  { $2 :: $1 }
  | includ { [$1] }

includ:
  | INCLUDE STRING SEMI { FileIncl($2) }
```

```
decls:
   /* nothing */ { [], [] }
 | stmt decls { ($1 :: fst $2), snd $2 }
 | tdecl_or_fdecl decls { fst $2, ($1 :: snd $2) }

tdecl_or_fdecl:
    fdecl { $1 }
  | tdecl { $1 }

fdecl:
  FUN ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
     { Func({ fname = $2;
         formals = $4;
         locals = [];
         body = List.rev $7 }) }

tdecl:
  TUPLE TID LBRACE tup_item_list RBRACE { Tup($2, List.rev $4) }

tup_item_list:
    tup_item                   { [$1] }
  | tup_item_list COMMA tup_item { $3 :: $1 }

tup_item:
    ID      { (String, $1) }
  | ID COLON tup_typ { ($3, $1) }

tup_typ:
    INTEGER   { Int }
  | REAL      { Float }
  | TEXT      { String }

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

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

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

stmt:
    expr SEMI { Expr $1 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR ID IN expr LBRACE stmt_list RBRACE { For($2, $4, Block($6)) }
  | FOR ID IN expr COLON stmt { For($2, $4, $6) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | BREAK SEMI { Break }
```

```
  | CONTINUE SEMI { Continue }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }


obj:
    ID                { Id($1) }
  | obj LBRACK expr RBRACK { Brac($1,$3,false) }

appended_obj:
    obj                { $1 }
  | obj DOLLAR ID      { Attr($1,$3) }
  | obj DOLLAR LPAREN expr RPAREN     { AttrInx($1,$4) }
  | obj LBRACK expr_opt COLON expr_opt RBRACK { Brac2($1,$3,$5) }
  | obj LBRACK RBRACK { Brac($1,Noexpr,false) }

lhs_appended_obj:
    appended_obj       { $1 }
  | AT obj LBRACK expr RBRACK { Brac($2,$4,true) }


/* | obj DOLLAR LPAREN LITERAL RPAREN     { AttrInx($1, Literal($4) ) } */


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


expr:
    LITERAL           { Literal($1) }
  | FLOAT             { FloatLit($1) }
  | STRING            { StrLit($1) }
  | TRUE              { BoolLit(true) }
  | FALSE             { BoolLit(false) }
  | appended_obj      { Obj($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr          { Unop(Not, $2) }
  | lhs_appended_obj ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | TID { TupInst($1) }
```

```
  | TID LBRACK RBRACK { TabInst($1) }
  | LBRACK actuals_opt RBRACK { Arr($2) }
  | LPAREN expr RPAREN { $2 }

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

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

23 src/scanner.mll

```
(* Ocamllex scanner for DaMPL *)

{ open Parser }


let digit = ['0'-'9']
let exponent = ['e' 'E']['+' '-']?digit+
let decimal = (digit+"."digit*)|(digit*"."digit+)

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '['      { LBRACK }
| ']'      { RBRACK }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ':'      { COLON }
| ';'      { SEMI }
| ','      { COMMA }
| '@'      { AT }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }



| "true"   { TRUE }
| "false"  { FALSE }


| "include"  { INCLUDE }
| "tuple"    { TUPLE }
```

```
| "$"        { DOLLAR }
| "break"    { BREAK }
| "continue" { CONTINUE }
| "fun"      { FUN }
| "in"       { IN }

| "real"        { REAL }
| "integer"  { INTEGER }
| "text"        { TEXT }

| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| (decimal exponent?)|(digit+exponent) as lxm { FLOAT(float_of_string lxm) }
| '"'([^'"']|("\\\""))*'"' as lxm { STRING(lxm) }
| ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| ['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { TID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal␣character␣" ^ Char.escaped char)) }

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

24 src/sconv.ml

```
open Ast
open Semt
open Stack
module StringMap = Map.Make(String);;
open Builtin
open Codegen

let param_separator = "__"

let fun_decls = ref StringMap.empty;;

let fun_abodies = ref StringMap.empty;;

let globals = ref (StringMap.singleton "args" (Array(String)) );;

let funs_to_reparse = ref [];;

let fun_parser_stack = (Stack.create () : string Stack.t);;

let loop_stack = (Stack.create () : int Stack.t);;

let tup_decls = ref StringMap.empty;;

let tup_attrs = ref StringMap.empty;;

let tup_inx = ref StringMap.empty;;

let tup_sizes = ref StringMap.empty;;

(* let tup_sizes = ref StringMap.empty;; *)

let rec string_of_param_typ = function
    Bool -> "bool"
  | Int -> "int"
  | Float -> "float"
  | String -> "str"
  | Void -> "void"
  | Array(x) -> "arr"^(string_of_param_typ x)
  | Tuple(s) -> "tup"^s
  | Table(s) -> "tab"^s
  | Undefined -> raise(Failure("Undefined param on string_of_param_typ"))

let rec string_of_typ = function
    Bool -> "Bool"
  | Int -> "Int"
  | Float -> "Float"
  | Void -> "Void"
  | String -> "String"
  | Array(x) -> "Array("^(string_of_typ x)^")"
  | Table(x) -> "Table("^x^")"
```

```
  | Tuple(x) -> "Tuple("^x^")"
  | Undefined -> "Undefined"


let get_built_in_name_folder map fun_prot =
  StringMap.add (get_built_in_name fun_prot) true map;;

let built_in_decls =
  List.fold_left get_built_in_name_folder StringMap.empty built_in_prototypes
      ;;

let generate_fun_name (name : string) (typs: typ list) : string =
  name^param_separator^(String.concat "_" (List.map string_of_param_typ typs)
      )

let built_in_prot_map_folder map fun_prot =
  let build_formal ftyp = (ftyp,"x") in
  let build_formals typlist = List.map build_formal typlist in
  let rec build_prot_map rtyp name protlist mmap = match protlist with
      prot::l ->
        let funname = generate_fun_name name prot in
        let semfdecl = {
          rtyp = rtyp; semfname = funname;
          originalname = name; semformals = build_formals prot;
          semlocals = ref StringMap.empty; sembody = [];
        } in StringMap.add funname semfdecl (build_prot_map rtyp name l mmap)
    | [] -> mmap
  in build_prot_map (get_built_in_rtyp fun_prot) (get_built_in_name fun_prot)
    (get_built_in_formals fun_prot) map;;


let parsed_funs = ref
  (List.fold_left built_in_prot_map_folder StringMap.empty
      built_in_prototypes);;

let is_logical_op = function
  | And | Or -> true
  | _ -> false

let is_comp_op = function
  | Equal | Neq | Less | Leq | Greater | Geq -> true
  | _ -> false

let pv k v = print_string("\nvar:␣"^k);;

let get_id_typ_from_locals_or_globals id fname =
  let semfdecl = StringMap.find fname !parsed_funs in
  if StringMap.mem id !(semfdecl.semlocals)
  then StringMap.find id !(semfdecl.semlocals)
  else if StringMap.mem id !globals
  then StringMap.find id !globals
  else raise(Failure("var␣"^id^"␣not␣found"))
```

```
let rec get_string_of_sem_obj semobj = match semobj with
    SId(id) -> id
  | SBrac(o,_,_,_) -> (get_string_of_sem_obj o)^"*"
  | SBrac2(o,_,_,_) -> (get_string_of_sem_obj o)
  | SAttr(_,_,o,attr,_) -> (get_string_of_sem_obj o)^"$"^attr^"$"
  | SAttrInx(_,_,o,_) -> (get_string_of_sem_obj o)^"$$"
  (* | _ -> raise(Failure("get_string_of_sem_obj case not implemented")) *)

let is_collection_access (obj : sem_obj) : bool =
  match obj with
    SBrac(_) -> true
  | _ -> false

let is_collection_range_access (obj : sem_obj) : bool =
  match obj with
    SBrac2(_) -> true
  | _ -> false

let is_array (typ : typ) : bool =
  match typ with
    Array(_) -> true
  | _ -> false

let is_tuple (typ : typ) : bool =
  match typ with
    Tuple(_) -> true
  | _ -> false

let get_tuple_name (typ : typ) : string =
  match typ with
    Tuple(t) -> t
  | _ -> ""

let rec is_array_of_undefined typ = match typ with
    Array(t) -> is_array_of_undefined t
  | Undefined -> true
  | _ -> false

let rec get_obj_typ (o : sem_obj) : typ = match o with
    SId(id) -> (* print_string(id); print_string(Stack.top fun_parser_stack);
         *)
      let fname = (Stack.top fun_parser_stack) in
      if fname = "_global_"
      then if StringMap.mem id !globals
        then StringMap.find id !globals
        else raise(Failure("global "^id^" not found"))
      else get_id_typ_from_locals_or_globals id fname
  | SBrac(o,e,i,_) ->
      let otyp = get_obj_typ o in
      ( match otyp with
        Table(t) -> Tuple(t)
      | Array(t) -> t
      | _ -> raise(Failure("cant get elem of non collection object"))
```

```
    )
  | SBrac2(o,_,_,_) ->
      let otyp = get_obj_typ o in
      ( match otyp with
        Table(x) -> Table(x)
      | Array(x) -> Array(x)
      | _ -> raise(Failure("cant get elem of non collection object"))
    )
  | SAttr(_,t,o,attr,_) -> t
  | SAttrInx(_,t,o,attr) -> t
(*
      let otyp = get_obj_typ o in (
      match otyp with
        Table(x) ->
          let attrId = x^"$"^attr in
          if StringMap.mem attrId !tup_decls
          then Array(StringMap.find attrId !tup_decls)
          else raise(Failure("tuple "^x^" has no attr "^attr))
      | Tuple(x) ->
          let attrId = x^"$"^attr in
          if StringMap.mem attrId !tup_decls
          then StringMap.find attrId !tup_decls
          else raise(Failure("tuple "^x^" has no attr "^attr))
      | _ -> raise(Failure("cant get attr of item that is not tuple or table
          "))
    )
  *)
  (* | _ -> raise(Failure("get_obj_typ case not implemented")) *)

let get_expr_typ (exp : sem_expr) : typ = match exp with (* TODO *)
    SLiteral(_) -> Int
  | SStrLit(_) -> String
  | SFloatLit(_) -> Float
  | SBoolLit(_) -> Bool
  | SObj(o) -> get_obj_typ o
  | SBinop(t,_,_,_) -> t
  | SUnop(t,_,_) -> t
  | SCall(name,_) -> (StringMap.find name !parsed_funs).rtyp
  | STupInst(tupname,_) -> Tuple(tupname)
  | STabInst(tupname) -> Table(tupname)
  | SArr(t,_) -> Array(t)
  | SSpecialCall(t,_,_) -> t
  | SNoexpr | _ -> Void
;;

(* TODO: improve this function *)
let rec get_typ_from_fun_sembody name body = match body with
    SReturn(exp)::l ->
      let rettyp = get_expr_typ exp in
      if rettyp == Undefined then
      get_typ_from_fun_sembody name l
      else if rettyp == Void then
      raise(Failure("fun "^name^": cant return void"))
```

```
      else rettyp
  | _::l -> get_typ_from_fun_sembody name l
  | [] -> Void;;

let rec validate_param_typs formals typs = match formals, typs with
   f::fl, t::tl -> ( (fst f) == t) && validate_param_typs fl tl
  | [], _::_ -> false
  | _::_, [] -> false
  | [], [] -> true;;

let rec generate_new_formals_and_locals oldformals typs =
  match oldformals, typs with
    oldformal::llst, t::tl ->
      let recval = generate_new_formals_and_locals llst tl in
      ( (t,snd oldformal)::(fst recval), StringMap.add (snd oldformal) t (snd
           recval) )
  | [], [] -> ([], StringMap.empty)
  | _, _ -> raise(Failure ( "generate_new_formals_and_locals␣error" ))
;;

let update_formal_typs_in_semfdecl (newname: string) (old : sem_func_decl) (
    typs : typ list) : sem_func_decl =
  let newformals, newlocals = generate_new_formals_and_locals old.semformals
      typs in
  { old with
    semfname = newname;
    semformals = newformals;
    semlocals = ref newlocals;
  }

let rec check_if_exists_in_stack stack str =
  if Stack.is_empty stack
  then false
  else
    if (Stack.pop stack) == str
    then true
    else check_if_exists_in_stack stack str;;

let rec add_id_to_map_backtrace mapref semobj expr_typ =
  (* print_string("backtrace adding "^(get_string_of_sem_obj semobj)^" "^(
      string_of_typ expr_typ)^"\n"); *)
  mapref := StringMap.add (get_string_of_sem_obj semobj) expr_typ !mapref;
  ( match semobj with
      SBrac(o,_,_,_)  -> (add_id_to_map_backtrace mapref o (Array(expr_typ)))
    | _ -> ()
  );
  ()

let rec add_id_to_map_helper mapref semobj expr_typ backtrace =
  (* print_string("adding "^(get_string_of_sem_obj semobj)^" "^(string_of_typ
       expr_typ)^"\n"); *)
  (if backtrace then ( match semobj with
      SBrac(o,_,_,_) -> (add_id_to_map_backtrace mapref o (Array(expr_typ)))
```

```
    | SBrac2(o,_,_,_) -> (add_id_to_map_backtrace mapref o (expr_typ))
    | _ -> ()
  ) else ());
  ( match expr_typ with
      Tuple(x) -> (
        let register_attr attr =
          ((mapref := StringMap.add
            ((get_string_of_sem_obj semobj)^"$"^(snd attr)^"$" ) (fst attr)
          !mapref);
          (* print_string("adding-attr "^((get_string_of_sem_obj semobj)^"$
              "^(snd attr)^"$" )^" "^(string_of_typ (fst attr))^"\n"); *)
          ())
           in
        List.iter register_attr (StringMap.find x !tup_attrs)
      )
    | Table(x) ->
      (add_id_to_map_helper mapref (SBrac(semobj,SNoexpr,false,(Tuple(x))) )
          (Tuple(x)) false);
      (let register_attr attr =
          ((mapref := StringMap.add
            ((get_string_of_sem_obj semobj)^"$"^(snd attr)^"$" ) (Array(fst
                attr))
          !mapref);
          (* print_string("adding-attr "^((get_string_of_sem_obj semobj)^"$
              "^(snd attr)^"$" )^" "^(string_of_typ (Array(fst attr)))^"\n");
               *)
          ())
           in
        List.iter register_attr (StringMap.find x !tup_attrs));
      ()
    | Array(x) -> (add_id_to_map_helper mapref (SBrac(semobj,SNoexpr,false,x)
        ) x false); ()
    | _ -> ()
  );
  (mapref := StringMap.add (get_string_of_sem_obj semobj) expr_typ !mapref);
  (* (StringMap.iter pv !mapref); *)
  ()

let add_id_to_map mapref semobj expr_typ =
  add_id_to_map_helper mapref semobj expr_typ true

let rec parse_semfdecl newname semfdecl typs =
  let callername = Stack.top fun_parser_stack in
  if check_if_exists_in_stack (Stack.copy fun_parser_stack) semfdecl.semfname
  then (* function already being parsed, so reparse caller later *)
    (funs_to_reparse := callername :: (!funs_to_reparse) )
  else (
    Stack.push newname fun_parser_stack;
    Stack.push 0 loop_stack;
    let newsemfdecl = update_formal_typs_in_semfdecl newname semfdecl typs in
    (* the part below is copied to check_if_fun_needs_reparse *)
    (* so any changes made here must be also done there *)
    let abody = StringMap.find semfdecl.semfname !fun_abodies in
```

```
    parsed_funs := StringMap.add newsemfdecl.semfname newsemfdecl !
        parsed_funs;
    let newsembody = convert_stmts abody in
    let newrtype = get_typ_from_fun_sembody newsemfdecl.semfname newsembody
        in
    if newrtype == Undefined then (
    ignore(Stack.pop fun_parser_stack);
    ignore(Stack.pop loop_stack);
    (funs_to_reparse := (newsemfdecl.semfname) :: (!funs_to_reparse) )
    ) else (
      let newnewsemfdecl = { newsemfdecl with
        sembody = newsembody;
        rtyp = newrtype;
      } in
      parsed_funs := StringMap.add newnewsemfdecl.semfname newnewsemfdecl !
          parsed_funs;
      ignore(Stack.pop fun_parser_stack);
      ()
    )
  )


and handle_scall (name : string) (typs : typ list) : string =
  let newname = generate_fun_name name typs in (
    if (StringMap.mem newname !parsed_funs)
    then ()
    else (
      if (StringMap.mem name built_in_decls) then
      raise(Failure("builtin␣fun␣"^name^"␣does␣not␣accept␣such␣params"))
      else
        if StringMap.mem name !fun_decls then (
          let semfdecl = StringMap.find name !fun_decls in
          if ( (List.length typs) == (List.length semfdecl.semformals) )
          then parse_semfdecl newname semfdecl typs
          else raise (Failure ("fun␣"^name^"␣call␣has␣wrong␣number␣of␣param")
              )
        ) else raise( Failure ("fun␣"^name^"␣used␣but␣never␣declared"))
    )
  ); newname

and convert_obj (o : obj) : sem_obj =
  convert_obj_checking_side o false

and convert_obj_checking_side (o : obj) (is_lhs : bool ) : sem_obj = match o
    with
    Id(id) -> SId(id)
  | Brac(o,e,i) ->
      let semo = convert_obj o in
      let seme = convert_expr e in
      let otyp = get_obj_typ semo in
      let etyp = get_expr_typ seme in
      (* (print_string ((string_of_typ otyp)^" ")); *)
      (* (print_string ((get_string_of_sem_obj semo)^"\n")); *)
```

```
    if etyp = Int || (seme = SNoexpr && is_lhs) then
      match otyp with
        Table(t) -> SBrac(semo,seme,i,Tuple(t))
      | Array(t) -> SBrac(semo,seme,i,t)
      | _ -> raise(Failure("cant␣get␣elem␣of␣non␣collection␣object"))
    else if (seme = SNoexpr) then
      raise(Failure("illegal␣empty␣index␣on␣rhs"))
    else raise(Failure("index␣must␣be␣integer"))
| Brac2(o,e1,e2) ->
    let semo = convert_obj o in
    let seme1 = convert_expr e1 in
    let seme2 = convert_expr e2 in
    let otyp = get_obj_typ semo in
    let e1typ = get_expr_typ seme1 in
    let e2typ = get_expr_typ seme2 in
    if (e1typ = Int || seme1 = SNoexpr) && (e2typ = Int || seme2 = SNoexpr)
         then
      match otyp with
        Table(_) | Array(_) -> SBrac2(semo,seme1,seme2,otyp)
      | _ -> raise(Failure("cant␣get␣elem␣of␣non␣collection␣object"))
    else raise(Failure("index␣must␣be␣integer"))
| Attr(o,attr) ->
    let semo = convert_obj o in
    let otyp = get_obj_typ semo in (
    match otyp with
      Table(x) ->
        let t =
          if StringMap.mem (x^"$"^attr) !tup_decls
          then StringMap.find (x^"$"^attr) !tup_decls
          else raise(Failure(x^"$"^attr^"␣doesnt␣exist")) in
        let inx = (StringMap.find (x^"$"^attr) !tup_inx)
        in SAttr(Table(x),Array(t),semo,attr,inx)
    | Tuple(x) ->
        let t =
          if StringMap.mem (x^"$"^attr) !tup_decls
          then StringMap.find (x^"$"^attr) !tup_decls
          else raise(Failure(x^"$"^attr^"␣doesnt␣exist")) in
        let inx = StringMap.find (x^"$"^attr) !tup_inx
        in SAttr(Tuple(x),t,semo,attr,inx)
    | _ -> raise(Failure("cant␣get␣attr␣of␣item␣that␣is␣not␣tuple␣or␣table"
        ))
  )
| AttrInx(o,expr) ->
    let semexpr = convert_expr expr in
    let etyp = get_expr_typ semexpr in
    let semo = convert_obj o in
    let otyp = get_obj_typ semo in
    if etyp = Int
    then (
      match otyp with
        Table(x) -> SAttrInx(Table(x),Array(String),semo,semexpr)
      | Tuple(x) -> SAttrInx(Tuple(x),String,semo,semexpr)
      | _ -> raise(Failure("cant␣get␣attr␣of␣item␣that␣is␣not␣tuple␣or␣
```

```
            table"))
      ) else raise(Failure("attribute␣index␣must␣be␣integer"))
  (* | _ -> raise(Failure("convert_obj case not implemented")) *)

(*
and str_of_obj = function
Id(s) -> s
*)

and convert_expr (exp : expr) : sem_expr = match exp with
    Literal(i) -> SLiteral(i)
  | StrLit(s) ->  SStrLit(s)
  | FloatLit(f) -> SFloatLit(f)
  | BoolLit(b) -> SBoolLit(b)
  | Assign(o,e) -> convert_assign o e
  | Binop(e1,op,e2) ->
    let seme1 = convert_expr e1 in let seme2 = convert_expr e2 in
    let e1typ = get_expr_typ seme1 in let e2typ = get_expr_typ seme2 in
    if (is_array e1typ) || (is_array e2typ) then
    ( if op = Add then
        if e1typ = e2typ then
        SBinop(e1typ,seme1,op,seme2)
        else raise(Failure("array␣concat␣operands␣must␣have␣same␣type"))
      else raise(Failure("invalid␣array␣operation"))
    )
    else if is_logical_op op then
    ( if ((e1typ = Bool || e1typ = Int) && (e2typ = Bool || e2typ = Int))
      then SBinop(Bool,seme1,op,seme2)
      else raise(Failure("invalid␣type␣on␣logical␣operation"))
    )
    else if is_comp_op op then
    ( if (e1typ = e2typ) || ((e1typ = Float || e1typ = Int) && (e2typ = Float
         || e2typ = Int))
      then SBinop(Bool,seme1,op,seme2)
      else if (e1typ = Float && e2typ = String)
      then SBinop(Bool,seme1,op,SCall("float__str",[seme2]))
      else if (e1typ = Int && e2typ = String)
      then SBinop(Bool,seme1,op,SCall("int__str",[seme2]))
      else if (e1typ = String && e2typ = Float)
      then SBinop(Bool,SCall("float__str",[seme1]),op,seme2)
      else if (e1typ = String && e2typ = Int)
      then SBinop(Bool,SCall("float__str",[seme1]),op,seme2)
      else raise(Failure("invalid␣type␣on␣comparision␣operation"))
    )
    else
      if e1typ = String then
        if e2typ = String && op = Add
        then SBinop(String,seme1,op,seme2)
        else raise(Failure("Invalid␣string␣operation"))
      else if e2typ = String then
        if e1typ = String && op = Add
        then SBinop(String,seme1,op,seme2)
        else raise(Failure("Invalid␣string␣operation"))
```

```
      else if e1typ = Bool || e2typ = Bool
        then raise(Failure("logical␣expression␣inside␣arithmetic␣expression")
            )
      else
        let resulttyp = if e1typ = Float || e2typ = Float
          then Float else Int in
        SBinop(resulttyp,seme1,op,seme2)
  | Unop(op,expr) ->
    let semexpr = convert_expr expr in
    let exprtyp = get_expr_typ semexpr in
    if op = Neg
      then if (exprtyp = Int || exprtyp = Bool)
        then SUnop(Bool,op,semexpr)
        else raise(Failure("neg␣unop␣on␣invalid␣type"))
    else
      if (exprtyp = Int || exprtyp = Float)
        then SUnop(exprtyp,op,semexpr)
        else raise(Failure("minus␣unop␣on␣invalid␣type"))
  | Call(s,lst) -> ( (* (print_string("_"^s^"_in "^(Stack.top
      fun_parser_stack)^"\n"); *)
          (* verifies if it is a special function *)
          let sbexpr = handle_special_function s lst in
          if sbexpr = SNoexpr then
            let exprs = convert_exprs lst in
            let typs = (List.map get_expr_typ exprs) in
            let funname = (handle_scall s typs ) in
            SCall(funname,exprs)
          else sbexpr )
  | Obj(o) -> SObj(convert_obj o)
  | TupInst(tup) -> (
      if StringMap.mem tup !tup_decls
      then STupInst(tup, (StringMap.find tup !tup_sizes))
      else raise(Failure("tuple"^tup^"␣never␣declared"))
    )
  | TabInst(tup) -> (
      if StringMap.mem tup !tup_decls
      then STabInst(tup)
      else raise(Failure("tuple"^tup^"␣never␣declared"))
    )
  | Arr(exps) ->
    let semexps = List.map convert_expr exps in
    let exptyps = List.map get_expr_typ semexps in
    let rec checktyps l = ( match l with
      [a] -> a
    | a::ll -> if (a = checktyps ll)
      then a else raise(Failure("type␣mismatch␣on␣array␣init"))
    | [] -> Undefined )
    in SArr(checktyps exptyps, semexps)
  | Noexpr -> SNoexpr


and convert_exprs exps = (List.map convert_expr exps)
```

```
and convert_stmt stmt = match stmt with
    Block(lst) -> SBlock(convert_stmts lst)
  | Expr(exp) -> (* print_string("E_in "^(Stack.top fun_parser_stack)^"\n");
      *)
      SExpr(convert_expr exp)
  | Return(exp) -> (* print_string("R_in "^(Stack.top fun_parser_stack)^"\n")
      ; *)
      if (Stack.top fun_parser_stack) <> "_global_" then
        SReturn(convert_expr exp)
      else raise(Failure("Illegal␣return␣outside␣function"))
  | If(exp,s1,s2) ->
      let semexpr = convert_expr exp in
      let typ = get_expr_typ semexpr in
      if typ = Bool || typ = Int then
        SIf(semexpr, convert_stmt s1, convert_stmt s2)
      else raise(Failure("invalid␣if␣condition"))
  | While(exp,stmt) ->
      (let loopcount = (Stack.pop loop_stack) in Stack.push (loopcount+1)
          loop_stack);
      let semexpr = convert_expr exp in
      let typ = get_expr_typ semexpr in
      if typ = Bool || typ = Int then (
        let ret = SWhile(semexpr, convert_stmt stmt) in
        (let loopcount = (Stack.pop loop_stack) in (Stack.push (loopcount-1)
            loop_stack));
        ret )
      else raise(Failure("invalid␣while␣condition"))
  | For(id,expr,stmt) -> (
      (let loopcount = (Stack.pop loop_stack) in Stack.push (loopcount+1)
          loop_stack);
      let semexpr = convert_expr expr in
      let typ = get_expr_typ semexpr in
      let value = match typ with
          Array(t) -> (
            if t = Undefined then raise(Failure("cant␣loop␣on␣array␣of␣
                undefined␣type"))
            else
              let mapref = if (Stack.top fun_parser_stack) = "_global_"
              then globals else (StringMap.find (Stack.top fun_parser_stack)
                  !parsed_funs).semlocals in
              let return = (mapref := (StringMap.add id t !mapref)); SFor(id,
                  t,semexpr, convert_stmt stmt) in
              (mapref := (StringMap.remove id !mapref)); return )
        | Table(t) -> (
            let mapref = if (Stack.top fun_parser_stack) = "_global_"
            then globals else (StringMap.find (Stack.top fun_parser_stack) !
                parsed_funs).semlocals in
            let return = (mapref := (StringMap.add id (Tuple(t)) !mapref));
                SFor(id,(Tuple(t)),semexpr, convert_stmt stmt) in
            (mapref := (StringMap.remove id !mapref)); return )
        | _ -> raise(Failure("cant␣make␣for␣on␣non␣iterable"))
       in (let loopcount = (Stack.pop loop_stack) in (Stack.push (loopcount
```

```
        -1) loop_stack)); value )
  | Break -> if (Stack.top loop_stack) > 0
    then SBreak else raise(Failure("cannot␣use␣break␣out␣of␣loop"))
  | Continue -> if (Stack.top loop_stack) > 0
    then SContinue else raise(Failure("cannot␣use␣continue␣out␣of␣loop"))
  (* | _ ->  raise( Failure ("convert_stmt case not implemented")) *)

and convert_stmts stmts = (List.map convert_stmt stmts)



and convert_assign o e = (
  (* print_string("converting "^(get_string_of_sem_obj (convert_obj o) )^"\n
     "); *)
  let semexpr = convert_expr e in
  let semobj = convert_obj_checking_side o true in
  let expr_typ = (get_expr_typ semexpr) in
  if (Stack.top fun_parser_stack) = "_global_"
  then (
    if StringMap.mem (get_string_of_sem_obj semobj) !globals
    (* outside function, global environment *)
    then (
      (* print_string((get_string_of_sem_obj semobj)^" "^(string_of_param_typ
          (get_obj_typ semobj))); *)
      (* print_string(" "^(string_of_param_typ(get_expr_typ semexpr))^"\n");
          *)
      let expectedtyp = StringMap.find (get_string_of_sem_obj semobj) !
          globals in
      if expectedtyp = get_expr_typ semexpr
        || (expectedtyp = Float && (get_expr_typ semexpr) = Int)
        || ( ( is_array expectedtyp ) && ( expr_typ = (Array(Undefined)) ) )
        then SAssign(expectedtyp,semobj, semexpr)
      else if ( (not (is_collection_access(semobj))) && expectedtyp = Int &&
          expr_typ = Float)
        then ( add_id_to_map globals semobj expr_typ;
        SAssign(Float,semobj, semexpr) )
      else if(is_collection_access(semobj) && expectedtyp = Undefined )
        then ( add_id_to_map globals semobj expr_typ;
        SAssign(expr_typ,semobj, semexpr) )
      else if(is_collection_range_access(semobj) && expectedtyp = (Array(
          Undefined)) && (is_array expr_typ))
        then ( add_id_to_map globals semobj expr_typ;
        SAssign(expr_typ,semobj, semexpr) )
      else if ( is_tuple expectedtyp && expr_typ = (Array(String)) ) then
        let tupname = (get_tuple_name expectedtyp) in
        let tupsize = StringMap.find tupname !tup_sizes in
        let params = [semexpr;SLiteral(tupsize);SString("map"^tupname)] in
        let newexp = (SSpecialCall(Tuple(tupname),"dampl_tup_convert",params)
            ) in
        SAssign(Tuple(tupname),semobj,newexp)
      else raise(Failure("cant␣change␣global␣"^(get_string_of_sem_obj semobj)
          ^"␣type"))
```

```
    ) else ( add_id_to_map globals semobj expr_typ;
    SAssign(expr_typ,semobj, semexpr)
    )
  ) else (
    (* inside function, local environment *)
    let semfdecl = StringMap.find (Stack.top fun_parser_stack) !parsed_funs
        in
    if StringMap.mem (get_string_of_sem_obj semobj) !(semfdecl.semlocals)
    (* check if it is a local variable *)
    then (
      let expectedtyp = StringMap.find (get_string_of_sem_obj semobj) !(
          semfdecl.semlocals) in
      if expectedtyp = get_expr_typ semexpr
        || (expectedtyp = Float && (get_expr_typ semexpr) = Int)
        || ( ( is_array expectedtyp ) && ( expr_typ = (Array(Undefined)) ) )
        then SAssign(expectedtyp,semobj, semexpr)
      else if ( (not (is_collection_access(semobj))) && expectedtyp = Int &&
          expr_typ = Float)
        then ( (add_id_to_map (semfdecl.semlocals) semobj expr_typ);
        SAssign(Float,semobj, semexpr) )
      else if(is_collection_access(semobj) && expectedtyp = Undefined )
        then ( add_id_to_map (semfdecl.semlocals) semobj expr_typ;
        SAssign(expr_typ,semobj, semexpr) )
      else if(is_collection_range_access(semobj) && expectedtyp = (Array(
          Undefined)) && (is_array expr_typ))
        then ( add_id_to_map (semfdecl.semlocals) semobj expr_typ;
        SAssign(expr_typ,semobj, semexpr) )
      else if ( is_tuple expectedtyp && expr_typ = (Array(String)) ) then
        let tupname = (get_tuple_name expectedtyp) in
        let tupsize = StringMap.find tupname !tup_sizes in
        let params = [semexpr;SLiteral(tupsize);SString("map"^tupname)] in
        let newexp = (SSpecialCall(Tuple(tupname),"dampl_tup_convert",params)
            ) in
        SAssign(Tuple(tupname),semobj,newexp)
      else raise(Failure("cant change var "^(get_string_of_sem_obj semobj)^"
          type"))
    ) else (
      (* check if it is a global *)
      if StringMap.mem (get_string_of_sem_obj semobj) !globals
      then (
        let expectedtyp = StringMap.find (get_string_of_sem_obj semobj) !
            globals in
       if expectedtyp = get_expr_typ semexpr
        || (expectedtyp = Float && (get_expr_typ semexpr) = Int)
        || ( ( is_array expectedtyp ) && ( expr_typ = (Array(Undefined)) ) )
          then SAssign(expectedtyp,semobj, semexpr)
        else if ( (not (is_collection_access(semobj))) && expectedtyp = Int
            && expr_typ = Float)
          then (add_id_to_map globals semobj expr_typ;
          SAssign(Float,semobj, semexpr) )
        else if(is_collection_access(semobj) && expectedtyp = Undefined )
          then ( add_id_to_map globals semobj expr_typ;
          SAssign(expr_typ,semobj, semexpr) )
```

```
        else if(is_collection_range_access(semobj) && expectedtyp = (Array(
            Undefined)) && (is_array expr_typ))
          then ( add_id_to_map globals semobj expr_typ;
          SAssign(expr_typ,semobj, semexpr) )
        else if ( is_tuple expectedtyp && expr_typ = (Array(String)) ) then
          let tupname = (get_tuple_name expectedtyp) in
          let tupsize = StringMap.find tupname !tup_sizes in
          let params = [semexpr;SLiteral(tupsize);SString("map"^tupname)] in
          let newexp = (SSpecialCall(Tuple(tupname),"dampl_tup_convert",
              params)) in
          SAssign(Tuple(tupname),semobj,newexp)
        else raise(Failure("cant change global "^(get_string_of_sem_obj
            semobj)^" type"))
        (* if not local or global, create new local var *)
      ) else ( add_id_to_map (semfdecl.semlocals) semobj expr_typ;
      SAssign(expr_typ,semobj, semexpr)
      )
    )
  )
)

and handle_special_function (name : string) (exprs : expr list) : sem_expr =
  match name with
  "len" -> ( match exprs with (* len call for arrays or tables *)
    [Obj(o)] -> ( match (get_obj_typ (convert_obj_checking_side o true)) with
          Table(_) -> SSpecialCall(Int,"dampl_arr_len",(convert_exprs exprs))
        | Array(_) -> SSpecialCall(Int,"dampl_arr_len",(convert_exprs exprs))
        | Tuple(name) -> (SLiteral(StringMap.find name !tup_sizes))
        | _ -> SNoexpr
      )
    | _ -> SNoexpr
    )
  | "print" -> (
      let rec get_array_dimension_and_type arr n = ( match arr with
          Array(t) -> get_array_dimension_and_type t (n+1)
        | t -> (n,t)
        ) in
      let semexprs = convert_exprs exprs in
      let typs = List.map get_expr_typ semexprs in
      match typs with
        [Array(t)] -> let dim,typ = get_array_dimension_and_type (Array(t)) 0
            in
          let tstr = Codegen.simple_string_of_typ typ in
          SSpecialCall(Void,"dampl_print_arr__"^tstr,semexprs@[(SLiteral(dim)
              )])
      | [Table(tname)] ->
          SSpecialCall(Void,"dampl_print_arr__tup",semexprs@[(SLiteral(1))])
      | [Tuple(name)] ->
          SSpecialCall(Void,"dampl_print__tup",semexprs)
      | _ -> SNoexpr
  )
  | "str" -> (
      let rec get_array_dimension_and_type arr n = ( match arr with
```

```
          Array(t) -> get_array_dimension_and_type t (n+1)
        | t -> (n,t)
        ) in
      let semexprs = convert_exprs exprs in
      let typs = List.map get_expr_typ semexprs in
      match typs with
        [Array(t)] -> let dim,typ = get_array_dimension_and_type (Array(t)) 0
            in
          let tstr = Codegen.simple_string_of_typ typ in
          SSpecialCall(String,"dampl_str_arr__"^tstr,semexprs@[(SLiteral(dim)
              )])
      | [Table(tname)] ->
          SSpecialCall(Void,"dampl_str_arr__tup",semexprs@[(SLiteral(1))])
      | [Tuple(name)] ->
          SSpecialCall(String,"dampl_str__tup",semexprs)
      | _ -> SNoexpr
  )

  | _ -> SNoexpr (* No special behavior *)

(*
and convert_stmt_ignoring_unparsed_call stmt = match stmt with
    Call(name,_) ->
      let semfdecl = StringMap.find name !fun_decls in
      if semfdecl.parsed
      then convert_stmt stmt
      else SIgnoredCall
  | _ -> convert_stmt stmt

  and convert_stmts_ignoring_unparsed_calls stmts = match stmts with
    s :: l -> (convert_stmt_ignoring_unparsed_call s) :: (
        convert_stmts_ignoring_unparsed_calls l)
  | [] -> [];;
*)

let convert_id_to_undef_typed_id (id : string) : typed_id = (Undefined, id);;

let rec convert_ids_to_undef_typed_ids (ids : string list) : typed_id list =
    match ids with
    id :: l -> (convert_id_to_undef_typed_id id) :: (
        convert_ids_to_undef_typed_ids l)
  | [] -> [];;

let create_fun_decl fd =
  (* let body = convert_stmts fd.body in
  let rettyp = (get_typ_from_fun_sembody fd.fname body) in *)
  fun_abodies := StringMap.add fd.fname fd.body !fun_abodies;
  let semfdecl = {
   rtyp = Undefined;
   semfname = fd.fname;
   originalname = fd.fname;
   semformals = (convert_ids_to_undef_typed_ids fd.formals);
   semlocals = ref StringMap.empty;
```

```
    sembody = []; } in
  fun_decls := StringMap.add fd.fname semfdecl !fun_decls;
  ();;

let rec create_funs_from_decls decls = match decls with
    Func(f_decl)::l -> create_fun_decl f_decl; create_funs_from_decls l
  | Tup(_)::l -> create_funs_from_decls l
  | [] -> ();;

let rec remove_formals_from_locals formals map = match formals with
  | f::lst -> StringMap.remove (snd f) (remove_formals_from_locals lst map)
  | [] -> map;;

let fun_map_to_list_fold_helper k v l =
  (* print_string ("function: "^(string_of_typ v.rtyp)^" "^v.semfname^"\n" );
      *)
  v.semlocals := remove_formals_from_locals v.semformals !(v.semlocals);
  v.semlocals := StringMap.filter
    (fun k -> (fun v -> (k.[(String.length k)-1] <> '*' && k.[(String.length
        k)-1] <> '$')))
    !(v.semlocals);
  if ( StringMap.mem v.originalname built_in_decls) then l else v::l;;

let globals_map_to_list_fold_helper k v l =
  if (k.[(String.length k)-1] = '*' || k.[(String.length k)-1] = '$')
    then l else (v,k)::l

(*
let reparse_fun2 name =
  Stack.push name fun_parser_stack;
  let semfdecl = StringMap.find name !fun_decls in
  let abody = StringMap.find name !fun_abodies in
  let newsembody = convert_stmts abody in
  let newrtype = get_typ_from_fun_sembody semfdecl.semfname newsembody in
  if newrtype == Undefined then
  raise(Failure("could not determine fun "^name^" return type"))
  else (
    let newnewsemfdecl = { semfdecl with
      sembody = newsembody;
      rtyp = newrtype;
    } in
    fun_decls := StringMap.add newnewsemfdecl.semfname newnewsemfdecl !
        fun_decls;
    ignore(Stack.pop fun_parser_stack);
    ()
  )
*)


let reparse_fun name =
  Stack.push name fun_parser_stack;
  let semfdecl = StringMap.find name !parsed_funs in
  let abody = StringMap.find semfdecl.originalname !fun_abodies in
```

```
  let newsembody = convert_stmts abody in
  let newrtype = get_typ_from_fun_sembody semfdecl.originalname newsembody in
  if newrtype == Undefined then (
  ignore(Stack.pop fun_parser_stack);
  raise(Failure("could not determine fun "^name^" return type"))
  ) else (
    let newnewsemfdecl = { semfdecl with
      sembody = newsembody;
      rtyp = newrtype;
    } in
    parsed_funs := StringMap.add newnewsemfdecl.semfname newnewsemfdecl !
        parsed_funs;
    ignore(Stack.pop fun_parser_stack);
    ()
  )

let create_tup_decl tdecl =
  let tsize = List.length (snd tdecl) in
  let create_tup_item inx item =
    (* print_string((string_of_int inx)^"declaring "^((fst tdecl)^"$"^(snd
        item))^"\n"); *)

    (tup_decls := StringMap.add ((fst tdecl)^"$"^(snd item)) (fst item) !
        tup_decls);
    (tup_decls := StringMap.add ((fst tdecl)^"$"^(string_of_int inx)) (fst
        item) !tup_decls);
    (tup_inx := StringMap.add ((fst tdecl)^"$"^(snd item)) inx !tup_inx);
    ()
  in
  (tup_sizes := StringMap.add (fst tdecl) tsize !tup_sizes);
  (tup_decls := StringMap.add (fst tdecl) (Tuple(fst tdecl)) !tup_decls);
  (tup_attrs := StringMap.add (fst tdecl) (snd tdecl) !tup_attrs);
  List.iteri create_tup_item (snd tdecl) ;;

let rec create_tups_from_decls decls = match decls with
    Func(_)::l -> create_tups_from_decls l
  | Tup(t)::l -> create_tup_decl t; t::(create_tups_from_decls l)
  | [] -> [];;


let convert (stmts, decls) =
  let tups = create_tups_from_decls decls in
  (Stack.push "_global_" fun_parser_stack);
  (Stack.push 0 loop_stack);
  create_funs_from_decls decls;
  let semstmts = convert_stmts stmts in
  (* (List.iter print_string !funs_to_reparse); *)
  (List.iter reparse_fun !funs_to_reparse);
  (
    (StringMap.fold globals_map_to_list_fold_helper !globals []),
    semstmts,
    (StringMap.fold fun_map_to_list_fold_helper !parsed_funs []),
    tups
```

```
);;
```

25 src/semt.ml

```
(* Semantic Tree *)

open Ast

module StringMap = Map.Make(String);;

type typed_id = typ * string

type sem_obj = (* lhs *)
    SId of string
  | SBrac of (* a[0] a[i] a[i+1] *)
    sem_obj * sem_expr *
    bool * (* indicate if it is insertion *)
    typ
  | SBrac2 of sem_obj * sem_expr * sem_expr * typ (* a[0:2] *)
  | SAttr of (* a$b *)
    typ * (* type of the object (tuple or table of something) *)
    typ * (* type of the attribute *)
    sem_obj *
    string * (* name of the attribute *)
    int (* index of the attribute*)
  (* e.g. in a$b 1st typ is typ of a and 2nd is typ of attr b*)
  | SAttrInx of (* a$(int_expr) - TREAT AS STRING *)
    typ * (* type of the object (tuple or table of something) *)
    typ * (* type of the attribute (String or Array(String) ) *)
    sem_obj *
    sem_expr (* index expression *)
and
 sem_expr =
    SLiteral of int
  | SBoolLit of bool
  | SFloatLit of float
  | SStrLit of string
  | SObj of sem_obj
  | SBinop of typ * sem_expr * op * sem_expr(* as on lhs *)
  | SUnop of typ * uop * sem_expr
  | SAssign of typ * sem_obj * sem_expr
  | SCall of string * sem_expr list
  | STupInst of string * int (* tuple instantiation *)
  | STabInst of string (* table instantiation e.g. Foo[] *)
  | SArr of typ * sem_expr list (* arrays e.g. [1,2,3] *)
  | SNoexpr
  (* The elements below are internal use only *)
  (* SpecialCall is an unchecked call, used for hardcode only *)
  | SSpecialCall of typ * string * sem_expr list
  | SString of string (* For use inside codegen only *)


type sem_stmt =
    SBlock of sem_stmt list
```

```
  | SExpr of sem_expr
  | SReturn of sem_expr
  | SIf of sem_expr * sem_stmt * sem_stmt
  | SFor of string * typ * sem_expr * sem_stmt (* for i in a *)
  | SWhile of sem_expr * sem_stmt
  | SBreak
  | SContinue

type sem_func_decl = {
    rtyp: typ;
    semfname : string;
    originalname: string;
    semformals : typed_id list;
    semlocals : typ StringMap.t ref;
    sembody : sem_stmt list;
  }


type sem_program =
  typed_id list *
  sem_stmt list *
  sem_func_decl list *
  tup list
```

26 src/tokenize.sh

```
ocamlc -c tokenizer.ml
ocamlc -o tokenizer parser.cmo scanner.cmo tokenizer.cmo
./tokenizer < $1 | menhir --interpret --interpret-show-cst parser.mly
```

27 src/tokenizer.ml

```
open Ast
open Parser


(*
let varArray = Array.make 26 0;;

let rec eval = function
    Lit(x) -> x
  | Var(x) -> varArray.(x)
  | Binop(e1, op, e2) ->
      (let v1 = eval e1 and v2 = eval e2 in
      match op with
             Add -> v1 + v2
      | Sub -> v1 - v2
      | Mul -> v1 * v2
      | Div -> v1 / v2 )
  | Asn(v,e) ->
    varArray.(v) <- eval e;
    varArray.(v)
  | Seq(e1,e2) ->
    ignore (eval e1);
    eval e2
*)




let string_of_token tk = match tk with
   SEMI -> "SEMI"
  | LPAREN -> "LPAREN"
  | RPAREN -> "RPAREN"
  | LBRACE -> "LBRACE"
  | RBRACE -> "RBRACE"
  | LBRACK -> "LBRACK"
  | RBRACK -> "RBRACK"
  | COMMA -> "COMMA"
  | COLON -> "COLON"
  | PLUS -> "PLUS"
  | MINUS -> "MINUS"
  | TIMES -> "TIMES"
  | DIVIDE -> "DIVIDE"
  | ASSIGN -> "ASSIGN"
  | NOT -> "NOT"
  | EQ -> "EQ"
  | NEQ -> "NEQ"
  | LT -> "LT"
  | LEQ -> "LEQ"
  | GT -> "GT"
  | GEQ -> "GEQ"
```

```
  | TRUE -> "TRUE"
  | FALSE -> "FALSE"
  | AND -> "AND"
  | OR -> "OR"
  | RETURN -> "RETURN"
  | IF -> "IF"
  | ELSE -> "ELSE"
  | FOR -> "FOR"
  | WHILE -> "WHILE"
  | INCLUDE -> "INCLUDE"
  | TUPLE -> "TUPLE"
  | DOLLAR -> "DOLLAR"
  | BREAK -> "BREAK"
  | CONTINUE -> "CONTINUE"
  | FUN -> "FUN"
  | IN -> "IN"
  | LITERAL(_) -> "LITERAL"
  | ID(_) -> "ID"
  | TID(_) -> "TID"
  | FLOAT(_) -> "FLOAT"
  | STRING(_) -> "STRING"
  | AT -> "AT"
  | EOF -> "EOF";;

let rec string_of_tokens tks = match tks with
  a::l -> (string_of_token a) ^ " " ^ (string_of_tokens l)
  | [] -> "";;

(*
let aa =
  let lexbuf = Lexing.from_channel stdin in
  let expr = Scanner.token lexbuf in
  let out = string_of_token expr in
  print_endline (out);;
*)

(* let a = *)
 (*
let paraporra = ref false;;

let lexbuf = Lexing.from_channel stdin in
while not (!paraporra) do
  let result = Scanner.token lexbuf in
    let out = string_of_token result in
    (ignore(paraporra := (out = "EOF" ));
    print_endline (out))
done

*)


  let lexbuf = Lexing.from_channel stdin in
  let rec loop acc =  function
```

```
    | EOF    ->  string_of_token EOF :: acc |> List.rev
    | x      ->  loop (string_of_token x :: acc) (Scanner.token lexbuf)
in
    loop [] (Scanner.token lexbuf)
    |> String.concat " "
    |> print_endline
```

28 testall.sh

```
#!/bin/sh

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

# Path to C Compiler (GCC)
GCC="gcc"
LDFLAG="-g␣-Llibs/"
CFLAG="-Ilibs/"
LDLIB="-ldampllib"
#GCC="usr/bin/gcc"

# DaMPL compiler.
DAMPLC="./dampl"

# Test files
files="tests/*.mpl"

Greetings() {
        echo ""
        echo "\t/*DaMPL␣Automated␣Test␣Script␣*/"
}

Build() {
        echo ""
        echo "Building␣the␣DaMPL␣Compiler␣..."
        cd src &&
        make &&
        cp dampl ../dampl
        cd ..
        cd libs &&
        make &&
        cd ..
        echo "Done."
}

Test() {
        if [ -s files_to_check.txt ]; then
                rm files_to_check.txt
        fi

        echo "These␣files␣presented␣different␣output␣than␣expected␣during␣the
            ␣automated␣test.\n" >> files_to_check.txt

        for file in $files
        do
                echo ""
```

```
            filename="${file%.*}"
            echo "File:␣\"${file}\""
            echo "Compiling␣from␣.mpl␣to␣.c␣..."
            eval "$DAMPLC" "${file}" ">" "${filename}.c" "2>␣./tests/temp
                .out"

            if [ -s ./tests/temp.out ]
            then
            echo "Compile␣time␣error␣detected␣..."

            diff -b ${filename}.out ./tests/temp.out > ${filename}.diff
                2>&1
                    rm ./tests/temp.out

                    if [ -s ${filename}.diff ]
                    then
                            echo "WARNING:␣Output␣differs␣from␣expected."
                            echo "${filename}\n" >> files_to_check.txt
                    fi

            echo "Done."
            continue
            else
                    echo "Generating␣the␣executable␣file␣..."
                    eval "$GCC" "-o" "${filename}" "${filename}.c" "
                        $CFLAG" "$LDFLAG" "$LDLIB"

                    echo "Testing␣the␣generated␣file␣..."
                    ./${filename} > ./tests/temp.out 2>&1

                    diff -b ${filename}.out ./tests/temp.out > ${filename
                        }.diff 2>&1

                    if [ -s ${filename}.diff ]
                    then
                            echo "WARNING:␣Output␣differs␣from␣expected."
                            echo "${filename}\n" >> files_to_check.txt
                    fi

                    rm ./tests/temp.out
                    rm ${filename}

                    echo "Done."
            fi
        done
}

# Main #

# Builds the compiler
Greetings
Build
Test
```

29 tests/array_to_tuple.mpl

```
tuple Tup {a, b:real , c:integer}

t = Tup;

a = ["hugo","3.65","22"];

t = a;

print(t);
```

30 tests/array_to_tuple.out

```
(hugo, 3.65, 22)
```

31 tests/conflicting_type.mpl

```
a = "134";
a = 3;
```

32 tests/conflicting_type.out

```
Fatal error: exception Failure("cant␣change␣global␣a␣type")
```

33 tests/fact.mpl

```
fun fact(n) {
        if( n < 0 )
                return -1;
        if( n < 2 ) {
                return 1;
        }
        return ( n * fact(n-1) );
}

print( fact(6) );
```

34 tests/fact.out

```
720
```

35 tests/files.mpl

```
str_file = readfile("./tests/text_input.txt");
print(str_file);

writefile("./tests/test.txt", "Bonjour␣Monde!");

file1 = open("./tests/text_input.txt", "r");
bonjour = readline(file1);

close(file1);
```

36 tests/files.out

```
Bonjour
Monde
Hello
World
Ola
Mundo
```

37 tests/for_loop.mpl

```
include "../DaMPL_stdlib.mpl";

array_real = [6.12, 3.1415, 2.21, 9.0001, 123.0, 5.6];

for i in array_real {
        print(str(i) + "\n");
}

tuple T {a:real, b:integer, c}

t = T;

t$a = 3.111;
t$b = 12356;
t$c = "abc";

for j in tuple_to_array(t) {
        print(str(j) + "\n");
}

for i in range(1, 11) {
        print(str(i) + "\n");
}
```

38 tests/for_loop.out

```
6.12
3.1415
2.21
9.0001
123
5.6
3.111
12356
abc
1
2
3
4
5
6
7
8
9
10
```

39 tests/helloworld.mpl

```
fun main() {
        print("Hello␣World!\n");
}

main();
```

40 tests/helloworld.out

```
Hello World!
```

41 tests/index_out_of_bounds.mpl

```
a = [1, 2, 3, 4, 5];

b = a[7];

print(b);
```

42 tests/index_out_of_bounds.out

```
Array out of bounds exception
```

43 tests/invert.mpl

```
fun invert(arr) {
        new_arr = [];

        i = len(arr) - 1;

        while(i >= 0) {
                new_arr[] = arr[i];
                i = i-1;
        }


        return new_arr;
}

a = [5, 4, 3, 2, 1];
a = invert(a);

print(a);
```

44 tests/invert.out

```
[ 1, 2, 3, 4, 5 ]
```

45 tests/join.mpl

```
fun join(arr, separator){
        i = 1;

        size = len(arr);

        new_str = str(arr[0]);

        while(i < size){
                new_str = new_str + separator + str(arr[i]);
                i = i + 1;
        }

        return new_str;
}

a = [1, 2, 3, 4, 5];
a_str = join(a, ",");

print(a_str);
```

46 tests/join.out

```
1,2,3,4,5
```

47 tests/print.mpl

```
tuple Test {a, b, c:integer}

t = Test;
t2 = Test;

t$a = "ta";
t$b = "tb";
t$c = 123;

t2$a = "t2a";
t2$b = "t2b";
t2$c = 456;

print(7);
print("\n");
print(2.67);
print("\n");
print("abc");
print("\n");
print(t);
print("\n");

arr_int = [1, 2, 3, 4, 5];
arr_float = [8.12, 9.123, 64.23, 3.1415];
arr_str = ["hello", "bonjour", "ola"];
arr_tup = [t2, t];

print(arr_int);
print(arr_float);
print(arr_str);
print(arr_tup);

arr_str2 = ["world", "monde", "mundo"];

arr_arr = [arr_str, arr_str2];

print(arr_arr);
```

48 tests/print.out

```
7
2.67
abc
(ta, tb, 123)
[ 1, 2, 3, 4, 5 ]
[ 8.12, 9.123, 64.23, 3.1415 ]
[ hello, bonjour, ola ]
[ (t2a, t2b, 456), (ta, tb, 123) ]
[ [ hello, bonjour, ola ]
[ world, monde, mundo ]
 ]
```

49 tests/print_void.mpl

```
include "../DaMPL_stdlib.mpl";

/* This function returns void */
fun print10Hellos() {
        for i in range(0, 10) {
                print("Hello\n");
        }
}

print( print10Hellos() );
```

50 tests/print_void.out

```
Fatal error: exception Failure("builtin␣fun␣print␣does␣not␣accept␣such␣params
    ")
```

51 tests/range.mpl

```
include "../DaMPL_stdlib.mpl";

a = 6;
b = 20;

c = range(a, b);

print(c);

print(range_s(a,b,3));
```

52 tests/range.out

```
[ 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 ]
[ 6, 9, 12, 15, 18 ]
```

53 tests/str_concat.mpl

```
a = "Hello";
b = "␣World";

fun scat(a,b) {
        return a+b;
}

print(scat(a,b));
```

54 tests/str_concat.out

```
Hello World
```

55 tests/str_conversion.mpl

```
a = 1234;
b = 3.1415;
c = [8, 9, 1, 10];
d = true;

print(str(a));
print("\n");

print(str(b));
print("\n");

print(str(c));
print("\n");

print(str(d));
print("\n");

tuple Tup {a:integer, b, c:real}

t = Tup;

t$a = 5;
t$b = "hello";
t$c = 3.131535;

print(str(t));
print("\n");

e = [1, 2, 3, 4, 5];

f = [c, e];

print(str(f));
print("\n");
```

56 tests/str_conversion.out

```
1234
3.1415
[ 8, 9, 1, 10 ]
true
(5, hello, 3.13154)
[ [ 8, 9, 1, 10 ][ 1, 2, 3, 4, 5 ] ]
```

57 tests/str_tuple.mpl

```
fun string_of_tuple(tup,separator) {
        str = tup$(0);

        i = 1;
        while(i < len(tup)) {
                str = str + separator + tup$(i) ;
                i = i+1;
        }

        return str;
}



tuple Person {name, age:integer, address}

b = Person;
b$name = "Bernie";
b$age = 21;
b$address = "Bauxita␣Street";

print(string_of_tuple(b, ",␣"));
```

58 tests/str_tuple.out

```
Bernie, 21, Bauxita Street
```

59 tests/test.txt

```
Bonjour Monde!
```

60 tests/test1.mpl

```
include "test2.mpl";

a=2;

fun saypingpong() {
        sayping();
        print("pong");
}

saypingpong();
```

61 tests/test1.out

```
pingpong
```

62 tests/test2.mpl

```
fun sayping() {
        print("ping");
}

a=1;
```

63 tests/test2.out

64 tests/test3.mpl

```
include "test1.mpl";

print(a);
```

65 tests/test3.out

```
pingpong2
```

66 tests/text_input.txt

```
Bonjour
Monde
Hello
World
Ola
Mundo
```

67 tests/undefined_function.mpl

```
fun giveMe2() {
        return 2;
}

a = 4;
b = 7;

print( giveMe2() + sum(a, b) );
```

68 tests/undefined_function.out

```
Fatal error: exception Failure("fun sum used but never declared")
```

69 tests/undefined_variables.mpl

```
a = 6;
b = 7;

c = a + b;

if( d == true ) {
        c = c - 10;
}
```

70 tests/undefined_variables.out

```
Fatal error: exception Failure("global d not found")
```