DAVE

Data Analysis (made) Very Easy

James HyunSeung Hong (hh2473)
Min Woo Kim (mk3351)
Fan Yang (fy2207)
Chen Yu (cy2415)

Professor Stephen A. Edwards
COMS W4115
Programming Languages and Translators
Fall 2015

Final Report
December 22, 2015

Contents

1 Introduction

The DAVE language is a programming language optimized for data retrieval, manipulation, and analysis. We designed DAVE with cross-dataset operations in mind, which are frequently needed but complicated to implement with existing technologies in today's data-intensive environments. Operators would be able to use DAVE to validate datasets, incorporate (parts of) datasets from different sources, split up oversized datasets, and conduct statistical analysis.

1.1 Motivation

Over the last few decades developers have created a variety of tools to help with data management and analysis. SQL and R language, for example, are two of the most prominent tools in this field, being widely utilized by business analysts, social scientists, statisticians, and many more.

Popular as they are, there still remain certain scenarios where both tools are imperfect. For instance, social scientists often need to incorporate data of distinct formations from different sources for their statistical analysis. SQL may work well for this job, yet it is designed for relational databases, a system which is not suitable for most statistical tasks. R language, on the other hand, is fully optimized for statistical analysis with all the functions and libraries, but it lacks the simplicity and usability of SQL in terms of managing data.

We have built DAVE to address this problem. DAVE is an integration rather than combination of the two prevailing languages, utilizing their advantages to build up a new world.

1.2 Target

The syntax of DAVE language is JAVA-like and simple enough for beginners to learn. The most unique and useful feature of DAVE is its three built-in data structures: **fld, rec, and tbl**, which represent a column of a table, a row of a table, and a table of data, respectively. These DAVE data structures will be particularly useful for cross-dataset operations. In addition, DAVE includes some built-in statistical functions to analyze the data of interest. By providing a generic tool that can be applied widely, DAVE envisions to develop a large open source community that is beneficial to all.

2 Language Tutorial

2.1 How the Compiler Works

2.1.1 Environment Setup

DAVE language needs **the GNU Compiler Collection 5** (GCC5) or later version to function, as it compiles to C++. The included helper scripts run only in an environment with **Bash,** and the **make** command would be useful when compiling your DAVE source code files. However, neither **Bash** nor **make** is required to run DAVE.

We recommend you to:

- Locate the DAVE Standard Library files (*dave.h*, *dave.hpp, dave_io.hpp*) under the same directory of your DAVE source codes
- Add the DAVE binaries to your **PATH** environment variable

2.1.2 Compiling and Running DAVE

The standard DAVE Package comes with the following source code files (under */src* folder):

ast.ml	compile.ml
dave_io.hpp	dave.hpp
dave.ml	dave.h
dave_core.dave	Makefile
Makefile.ocaml	parser.mly
sast.ml	scanner.ml
semanticAnalysis.ml	semanticExceptions.ml

You need to compile the provided source code files with DAVE compiler first before compiling and running any DAVE program. In order to make the DAVE compiler, run the following commands:

```
make
```

Suppose the source code file of your DAVE program is named *my_program.dave*, the following commands would compile your DAVE program to executable:

```
./dave -c <my_program.dave
g++ dave.cc
```

Alternatively, you can compile with the friendly shell script with the following command:

```
./ezcompile.sh
```

It is important to note that Makefile, generated compiler, and ezcompile.sh are all under /src directory. Therefore, the above command will work if you are currently under /src directory.

2.2 Remarkable Features

As mentioned in the introduction, the most remarkable feature of DAVE is that it provides the data structures representing a record (**rec**), a field (**fld**), and a table (**tbl**). **fld** contains a sequential collection of homogeneous variables. It is similar to an array data structure in other languages, but **fld** contains a label that can be used to easily retrieve or modify the data of your interest. Because its elements are homogeneous variables, most of the arithmetic operations can be applied to some parts or the whole of the dataset.

rcd contains a sequential collection of heterogeneous variables. Each tuple-like **rcd** variable contains a name and the corresponding contents. Each content variable can be of int, double, bool or string data types. Typically, a row of the table can be translated into a **rcd**. Arithmetic operations may be difficult to be applied here, but it would be useful for applying querying functions.

tbl represents a data table that consists of one or more flds or recs. Nearly every SQL-like dataset can be translated into a **tbl** variable. Every **tbl** can be decomposed into one or more flds or recs. It is particularly useful for the cross-functional data analytics process. As we provided the functionalities for users to directly generate a **tbl** variable from the given dataset, it would be very convenient for the user to handle multiple datasets simultaneously, incorporate the required information, apply analysis, and write the result back.

3 Reference Manual

The *DAVE Language Reference Manual* follows the organizational structure set by Dennis M. Ritchie in his *C Reference Manual*.

In this manual, syntactic categories are indicated by *italic* type, while literal words, characters and code examples are in written in consolas. Optional expressions are indicated by $_{opt}$.

3.1 Lexical Conventions

3.1.1 Character Set

DAVE supports a subset of ASCII characters, whose decimal codes in the ASCII table range from 32 ([SPACE]) to 126 (~). In other words, DAVE understands [SPACE], !, “ ,#, $, %, &, ’, (,), *, +, -, ., /, number 0-9, :, ;, <, =, >, ?, @, upper-case letter A-Z, lower-case letter a-z, [, \,], ^, _, \, {, |, }, ~. Escape sequences, such as \t, \n, and \r, are also included in DAVE’s character set.

3.1.2 Comment

A multi-line comment block starts with /* and ends with */. Also the single-line comment could be handled by the same rule. But the // for single-line comment is also acceptable. Comments are ignored by the syntax.

3.1.3 Identifiers

A legitimate identifier is a sequence of upper-case letters A-Z, lower-case letters a-z, numbers 0-9, and/or the underscore _. Additionally, the identifier must start with upper-case letters, lower-case letters. Identifiers are case-sensitive; “countryGDP” and “countrygdp” are two separate identifiers. Identifiers cannot equal to the reserved words (i.e. keywords), and each can be no more than 31 characters.

3.1.4 Reserved keywords

The following identifiers are reserved; they cannot be used as ordinary identifiers. Also, their spellings must be exactly same as follows:

int	float	bool	str	tbl
rec	fld	if	else if	else

return	for	while	break	continue
true	false	new		

3.1.5 Constants

DAVE programmers can declare primitive constants in four data types (whose details would be specified in latter sections): *integer* (int), *float* (float), *string* (str), and *boolean* (bool).

- ***Integer*** constants are decimal numbers consisting of a series of digits (0-9) and a leading sign character (+ or -). It is fine not to include the sign character; in this case the statement will always be parsed as a positive integer number.
- ***Float*** constants are decimal numbers with a decimal separator (.), a series of digits (0-9) before the separator (integer part), a series of digits after the separator (fraction part) and a sign character (+ or -). Similar to *integer constants*, the sign character could be omitted, rendering the constant positive automatically. Additionally, the integer part and the fraction part cannot be absent at the same; a digit of 0 would be automatically filled into the absence if the user decide not to declare the integer part or fraction part.
- ***Boolean*** constants can have either the value true or the value false. The values are case-sensitive; inputs like True, TRUE, False or FALSE are not Boolean values.
- ***String*** constants are a series of supported ASCII characters surrounded by a pair of quotation marks (“”). A backslash (\) is required for certain characters to be included in the string, such as:

Character	Input as	Name
\n	\n	Line Feed
\t	\t	Horizontal Tab
\r	\r	Carriage Return
”	\”	Quotation Mark
\	\\	Backslash

3.2 Conversions

There are number of operators that can cause conversion between different data types in our DAVE language.

3.2.1 Number conversions

int and float can be converted to each type by using float() and int() functions, respectively. When converting float to int, decimal values will be rounded down. For example:

```
int x = 1;              /* x = 1 */
float y = float(x);     /* y = 1.0 */
int z = int(y + 2.5);   /* z = 3 */
```

3.2.2 Boolean conversions

int, float, and str can be converted to a boolean value by using bool() function. x is converted to false, if x is none, empty or omitted, otherwise it returns true. For int and float, bool(x) returns true for x values of 0 or 0.0 and false otherwise. For example:

```
bool e = bool("");          /* e = false */
bool s = bool("0");         /* s = true */
bool z = bool(0.00);        /* z = false */
bool n = bool(none);        /* n = false */
```

3.2.3 fld conversions

int[], float[], bool[], and str[] can be converted to a fld by using new fld() function. A field name must be specified when converting, and the new before conversion function is compulsory. For example:

```
int[] num = [21, 23, 24, 26];
fld ages = new fld(num, "age");
```

3.2.4 tbl conversions

fld and rec can be converted to a tbl using tbl() function. We must construct the fld and rec array explicitly before using it as an argument in the tbl() function. For example, tbl a = new tbl([b, c]); will not work. Also, it is important to note that we must use the word "new" to instantiate a new tbl. One or more fld or rec instances can be converted into one tbl. When converting multiple flds into a tbl, their sizes must be equal in order to have same number of values for each record. When converting multiple rec's into a tbl, some length for each record is required, but their sequence could be disordered. For example:

```
fld a = new fld([20, 22, 24, 26], "age");
fld b = new fld(["James", "Min", "Chen", "Fan"], "name");
fld[] fld_array = [a, b];
```

```
tbl roster = new tbl(fld_array);     /* table with 4 records and 2 fields */
rec c = new rec(name: “Min Woo”, age: 22);
rec d = new rec(age: 99, name: “John”);
rec[] e = [c, d];
tbl roster2 = new tbl(e);            /* table with 2 records and 3 fields */
```

3.3 Expressions

Expressions in DAVE are a combination of identifiers, constants, function calls and operators.

3.3.1 Primary Expressions

Primary expressions involving function calls group left-to-right.

3.3.1.1 *identifier*

An identifier is a name used to identify an entity in a program. It is preceded by its type (such as str), and it should be suitably declared as explained at above (§3.1.3).

3.3.1.2 *constant*

A decimal, floating constant, string, or boolean is a primary constant. A decimal is of type int, floating constant is of type float, string are of type str, and boolean is of type bool.

3.3.1.3 (*expression*)

A parenthesized expression is simply an expression surrounded by a parenthesis; it is equal to an unadorned expression.

3.3.1.4 *primary-expression* ($expression\text{-}list_{opt}$)

A function call is a primary expression followed by an optional list of expressions in parenthesis. The list of expressions may be empty, a single expression, or a comma-separated list of expressions. A copy is made of each actual parameter passed to the function, so changes to its formal parameters by a function does not affect the values in the actual parameters.

3.3.1.5 *primary-expression* [*expression*]

A primary expression followed by an expression in square brackets is a primary expression. A primary expression must be an array of a primitive type (int[], float[], bool[], or str[]), and an expression must be an integer within the range of the array size.

An expression can also contain a colon (:), which has integers on the left and right. This specifies the range of primary expressions to return. When an integer is not specified on the right of the colon, the length of the array will be used as default. The resulting object is an array of primary expression. For example:

```
str[] names = [“James”, “Min Woo”, “Chen”, “Fan”];
str first = names[0];       /* first = “James” */
str[] rest = names[1:];     /* rest = [“Min Woo”, “Chen”, “Fan”] */
```

3.3.2 Unary Operators

All unary arithmetic operations have the same priority, and unary operators group right-to-left.

3.3.2.1 - *expression*

The unary - operator yields the negative of the expression. The type of expression must be int or float.

3.3.2.2 *! expression*

The unary ! operator yields the logical negation of the expression. The type of expression must be int or bool. The result of the expression is 1 or true if the value of the expression is 0 or false, 0 or false if the value of the expression is non-zero or true.

3.3.2.3 *expression* ++

The result of the expression is the value of the object referred to by the expression. After the result is returned, the expression is incremented by 1.

3.3.2.4 *expression* --

The result of the expression is the value of the object referred to by the expression. After the result is returned, the expression is decremented by 1.

3.3.3 Multiplicative operators

multiplicative operators group left-to-right.

3.3.3.1 *expression * expression*

The binary * operator indicates multiplication. If both operands are int, the result is int; if one operand is int and the other is float, the result is float; if both operands are float, the result is float.

3.3.3.2 *expression / expression*

The binary / operator indicates division. The types of operands and results are the same as multiplication.

3.3.3.3 *expression % expression*

The binary % operator indicates remainder of the division of the expressions. Both operands must be int, and the result is int.

3.3.4 Additive operators

Additive operators group left-to-right.

3.3.4.1 *expression + expression*

The result is a sum of the two expressions. If both operands are int, the result is int; if one operand is int and the other is float, the result is float; if both operands are float, the result is float; if both operands are str, the result is str. You can also use + operator to add two tables, will return a new tbl instance. No other type combinations are allowed.

3.3.4.2 *expression - expression*

The result is a difference of the two expressions. Unlike addition, the operands could not cover str or tbl type, only int and float are permitted. The remaining types of operands and the results are the same as addition.

3.3.5 Comparison operators

Comparison operators <, >, ==, >=, <=, and != compare the values of two objects and yield either true or false. The operators have the same priority. Comparison operators group left-to-right.

When the compared objects are not of the same type, If they are int or float, they will be converted to a common type. Otherwise, they could only applied to unequal and equal operation based on == or !=, and throw an error on any other comparison.
The objects of the same type are compared according to the following rules:

- Numbers are compared arithmetically.
- Strings are compared lexicographically based on their ASCII numeric equivalents. The comparison conforms to sequence of the string characters, each corresponding element pair at a time. If the whole characters the same, the return would be true, otherwise until the pair of character are different.

3.3.6Assignment operators

Assignment operators group right-to-left.

3.3.6.1 *expression = expression*

The value of the left expression is replaced by the expression on the right. The operands need to be of same type, and it can be either int, float, str, bool. The invoke of rec, fld and tbl need to invoke the initialization function.

3.3.6.2 *expression += expression*
3.3.6.3 *expression -= expression*
3.3.6.4 *expression *= expression*
3.3.6.5 *expression /= expression*
3.3.6.6 *expression %= expression*

The behavior of the expression of the form "E1 op= E2" category is equivalent to "E1 = E1 op E2".

3.4 Declarations

Before our usage of identifiers within the functions, we need to give a declaration in order to specify their interpretation. The declaration of identifier might contain the type-specifier part and the declarator-list part.
Like C and Java, the type-specifier of DAVE is compulsory. And the declarator-list could be just an identifier, or several composing a list.

declaration:
type-specifier declarator-list$_{opt}$

3.4.1 Type Specifier

The type-specifiers in DAVE might cover the kinds of data structures mentioned below. The declaration of rec is likely a combination of C language and Java language. Thus would be further illustrated later. The first line are primitive types while the second are unique ones in DAVE language

int	float	bool	str
tbl	rec	fld	

3.4.2 Declarators

The declarator-list could be a sequence of declarators separated with a single comma. The type of storage class of objects could be known due to its corresponding specifier.

declarator:
identifier
declarator(declarator-list$_{opt}$)
declarator[constant-expression$_{opt}$]

We might discuss more about the last two kinds of declarators. If the declarator has the form like D(), it would be treated as the type "function returning …", where the "…" is the type which the identifier had depending on the former type specifier. The declarator-list represents the parameter of that function.

If the declarator has the form like D[], or D[*constant-expression$_{opt}$*], the constant expression should be an integer determining the storage distribution. Such a declarator makes the identifier have the type "array". The array could be extended from an existing array, from a single-axis array to a two-dimensional one (matrix). Generally, we seldom utilize more than three-dimensional array in our DAVE applications.

Not all the possible combine situations above would be workable, especially for the ones linked with fld, tbl and rec. Returning array in the function or constituting array of function is not permitted. But returning fld, rec or tbl is somewhat accessible. However, only fld and rec accept array declarator, but not tbl.

As an example, the declaration

int i, f(), a[]

declares an integer i, a function f returning an integer, an array of integer numbers. The same happens for the primitive specifiers double, str and bool.

3.4.3 Field Declarations.

The declaration

fld fld1 = new fld([18, 21, 25, 34], "age");

declares a field fld1, the storage type differ from the right side of the equation, user might not need to declare that explicitly, the parser would recognize thus automatically. The informations before the comma are the data stored in the field and the information after the comma is the name of the field.

In this example, It indicates that the field1 contains four integer entities. The field may be just analogous to an one-dimensional array, except the implicit type definition. However, the most distinct between field and one-dimensional array is that it need to have a name, otherwise the corresponding upper layer table could not recognize it correctly. The conversion between one-dimensional array and field could be found in the Conversion Chapter. fld types also support array extension. But typically would be converted into tbl as more functions provided. Functions returning a field is also acceptable.

3.4.4 Record and Table Declarations

Since the data stored in the record possess different types of attribute, we need to point out the name of each entity (content), referring which field they would link with. The difference between our record and the traditional struct is that we do not need to indicate the type of the data, they might be recognize by the parser, however the name of each entity is indispensable.
The declaration

```
rec rec1 = new rec (name: “Fan Yang”, age: 22);
```

declares a record rec1, the entities are separated with commas. And the part before the colon is the name of entity. The function returning record would be accepted in DAVE. And fld types also support array extension. But typically would also be converted into tbl as for convenience.

The declaration of table type need to be done through the conversion, not to burden too much on the parser. The conversion between three types tbl, rec and fld could be referred from the Conversion Chapter, thus we would not duplicate here. Just gave an simple example of table declaration here.
The declaration

```
fld a = new fld([20, 22, 24, 26], “age”);
fld b = new fld([“James”, “Min”, “Chen”, “Fan”], “name”);
fld[] e = [a, b];
tbl roster = new tbl(e);                /* table with 4 records and 2 fields */
rec c = new rec(name: “Min Woo”, age: 22);
rec d = new rec(age: 99, name: “John”);
rec[] f = [c,d];
tbl roster2 = new tbl(f);               /* table with 2 records and 3 fields */
```

3.5 Simple Statements

Each simple statement is identified by a semicolon at the end. Several simple statements may also occur on a single line separated by semicolons.

3.5.1 Expression statements

Expression statements are used to evaluate and store a value, or to call a function. Any valid expressions that are comprised within a single logical line is a statement.
Here are some examples:

```
x++;
y = 2x + 5;
```

3.5.2 Assignment statements

Assignment statements are used to bind or rebind references to values and to modify attributes of mutable objects. The assignment value of the right must be consistent with the type of variable in the left. Here are some examples:

```
a[1] = “hello”; b = 4;
fld.length = 10;
```

3.5.3 print statement

print stringifies each expression and writes the resulting object to standard output. A newline character is written at the end if not specified otherwise. As for tbl, rec and fld data type, Multiple expressions can be written in a single line when concatenated. separated by the tab character. Typically, each line of a fld contain one element, and rec might have two lines, one for the names and another for the contents.Take an example of the print of table:

height	name	age	weight
178	tom	20	68
195	jacky	21	81
175	jay	22	77
60	janice	23	48
174	sam	24	69

3.5.4 save statement

save writes a formatted text, which represents and saves a DAVE table, to a specified path. The output format of the save format is just like the print format. The only difference is a semicolon to mark the end of line, convenient for load statement.
For example:

```
str path = “documents/roaster.txt”;
fld a = new fld([21, 23, 24, 26], “age”);
fld b = new fld([“James”, “Min”, “Chen”, “Fan”], “name”);
tbl table = new tbl([a, b]);
save(table, path);
```

Another example of the output format of save statement:

```
height name  age   weight      ;
178    tom   20    68    ;
195    jacky 21    81    ;
175    jay   22    77    ;
60     janice 23   48    ;
174    sam   24    69    ;
```

3.5.5 load statement

load validates the given text file, converts it into the tbl object and returns the object. If the user just wanna fetch part of the table, the built-in function of access would assist with you, but not include it here.
For example:

```
str path = “documents/roaster.txt”;
tbl table = load(path);
```

3.5.6 return statement

return terminates the current function call with the expression list, typically a single primitive constant or variable. For example:

```
return true;
return “Hello World!”;
```

If the function intended to return void, then the expression list could be absent. Otherwise it is compulsory.

3.5.7 break statement

The break statement terminates the loop without executing the rest of the code block.

3.5.8 continue statement

The continue statement skips the rest of the code block and triggers the next iteration.

3.6 Compound Statements

Compound statements enclose any number of statements within one or more clauses, comprised within a pair of bracket. Compound statements usually span multiple lines.

3.6.1 if statement

The if statement conditionally executes a set of statements, based on the truth value of a given expression. The sequencing else clause and else statement are optional. If else part not existed, the compiler would simply skip with a false condition result. If more than two branches are reachable, the if statement support multiple nesting.
The following is the general form of an if statement:

```
if (condition1) {
        statement1;
} else if (condition2){
        statement2;
} else {
        statement3;
}
```

If *condition1* evaluates to true, then *statement1* is executed and *statement2*, *statement3* are ignored. If *condition1* evaluates to false and *condition2* evaluates to true, *statement2* is executed and other twos are ignored. If neither *condition1* nor *condition2* is met, *statement3* is executed.

3.6.2 for statement

The for statement is used to iterate under certain condition. The condition part of it typically consist of three expressions, separated by the semicolons. Any or all of the three expressions are optional. However, if all are not set, the statement would be processed infinite times. After the each time of execution, the iteration worked, and checked whether has reached the end condition. If not, then would still carry on the circulation work.

The following is the general form of a for statement:

```
for (start condition; end condition; iteration) {
        block of code;
```

```
}
```

3.6.3 while statement

The while statement is used to execute a block of code as long as a specified condition is true (none-zero). The test takes place before each execution of the statement. If the initialize condition is not false, the code of the block would be neglected directly.
The following is the general form of a while statement:

```
while (condition) {
        block of code;
}
```

3.6.4 Function definitions

A user can define a function object with none to many abstract parameters. The execution of a function definition binds the function name to a function object. The function body is only executed when the function is called.
The following is the general form of a int function definition and the example of calling the function, in DAVE, primitive types as well as fld, rec, tbl could the the return type:

```
int func_name(parameters) {
        function body;
        return statement;
}
int a = func_name(parameters);
```

3.7 Scope

DAVE defines code blocks by a pair of brackets. A variable declared in a certain code block cannot be referenced outside the code block. Variables that are only referenced within a code block are implicitly global. Variables declared within a function body are local to the function body, unless explicitly declared as global.

4 Project Plan

4.1 Planning Process

During the planning process, we discussed what kind of language we want to implement, several ideas were proposed, but considering the viability and the applicability, we determined that the language of data analytical would be a wise choice and named it as DAVE, a quite decent and easy to remember name. At first, we also considered the data visualization as part of our language. However, due to the lack of time during the final weeks, the visualization part was not implemented, which is slightly different from our original imagination.

4.2 Development Process

After we had decided the general of the language and its purpose, we have set several specified milestone for the language development. We met once a week together, typically Friday noon, to discuss the progress of our project and distribute task in the sequencing week. Also, after midterm, we met with the TA Prachi approximately once two weeks, typically Monday or Tuesday noon to report our current situation to her. During our DAVE project, although we have confronted such the bottleneck, but with the endeavor of our teammates and the guidance of the TA, these difficulties were solved one by one and our project is realized ultimately.

We started our work from the frontend, following the stage of the compiler architecture. The lexer and parser were finished first, and then the semantic check and the code generation were done. At the same time, we constructed the basic C++ library to implement the fundamental data types and functions for DAVE.

4.3 Test Process

Since the code generation part of our language fulfilled all its functionality a little late, the test progress seems to have only limited time. However, the test is undertaking just along with the compiler programming. We utilize proper test case to examine the compiler once an module is completed. To check it part by part would be more wise since the workload of debugging the whole compiler at once would be so tremendous. Smaller test cases seeming trivial are critical here, the further specification of test would be illustrated in the Chapter Six, Test Plan.

4.4 Programming Style Guide[1]

- Indentations comprises of four spaces, and are implemented with spaces instead of tabs.
- Method names are followed immediately by a left parenthesis.
- Array deferences are followed immediately by a left sure bracket.
- Binary operators should have a space on either side.
- Unary operators are followed/preceded immediately by their operand.
- if, while, for and switch are followed by a space.
- Avoid making a line too long.
- The braces should locate as the following:

```
type t_expr = {
(* Some Code Here *)
}
```

Instead of

```
type t_expr =
{
(* Some Code Here *)
}
```

- Adopt the *lowerCamelCase*.
- The first letter of each word in the name will be uppercase. All other letters will be in lowercase.
- Start the comment in a new line.
- Self-documenting code.
- Avoid long names in general.
- Avoid long parameter lists in general.

4.5 Timeline

Date and Time	Milestone
Sep. 30th	Language Proposal Completed
Oct. 26th	Language Reference Manual Completed
Oct. 28th - Nov. 9th	Scanner and Parser Completed
Nov. 10th - Nov. 15th	Minimally Working Build of DAVE Compiler Completed

[1] We referenced a JAVA Programming Style Guide (http://www.javaranch.com/style.jsp) for our own DAVE Programming Style Guide.

Nov. 16th	Hello_World Presentation
Nov. 17th - Dec. 14th	Expanding Features
Dec. 15th - Dec. 20th	Review and General Testing
Dec. 22nd	Final Report Completed

4.6 Roles and responsibilities

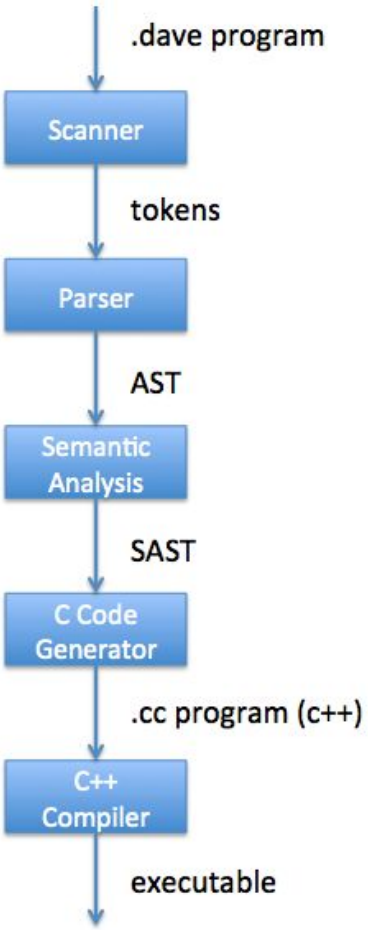
The professor suggested that each team member should be assigned a role of Manager, Tester, System Architect and Language Guru initially. However, it seems too stubborn to stick to it. Most of the works like language manual determine need the discuss of whole group, so in our DAVE project, the role of each group member is rather fluid. The major part of work of each team member is listed in the below sheet.

Name	UNI	Roles	Responsibilities
Hyun Seung Hong	hh2473	Manager	Coordinates within the team
Min Woo Kim	mk3351	Language Guru	Implementing the DAVE language
Fan Yang	fy2207	System Architect	Define the Architecture of the DAVE language
Chen Yu	cy2415	Tester	Testing the Components
	Some of the responsibilities are shared among members due to contingencies		

4.7 Software Development Environment

The DAVE project is developed on Mac OS X 10.10 (Yosemite) and Mac OS X 10.11 (El Capitan). Source codes and documents are controlled using Git and hosted on github.com. The compiler and related test cases is created in OCaml and completed with Sublime Text 3. Helper scripts are coded with TextEdit.

5 Architectural Design



The architectural design of DAVE is demonstrated in the block diagram above. It is split into 4 major sections: Scanner (scanner.mll), Parser (parser.mly), Semantic Analysis (semanticAnalysis.ml), and C++ Code generator (compile.ml).

5.1 Scanner

The scanner scans the input source program and produces the tokenized output. It ignores all white spaces, new lines, comments, spaces and tabs. It also reports an error whenever an invalid character is found in the program.

5.2 Parser & AST

Parser is used to parse the sequence of tokens that are generated by the scanner. After it scans the tokens, it generates an **Abstract Syntax Tree**. Parser also checks the syntax and reports an error if it catches any syntax errors.

5.3 Semantic Analysis & SAST

This step is used to parse the generated AST and conduct the semantic check by checking the types and semantics of every statement. In order to maintain variable and function references (v_context and f_context, respectively), we used a hash map with StringMap. After this step, a Semantically-checked Abstract Syntax Tree (SAST) is generated, which contains the type to use in the C code generation step.

5.4 C++ Code Generator

The C++ Code Generator takes the semantically-checked AST to generate a corresponding C code for each of the statement. Since all syntax and semantics have been checked using the parser and semantic analysis, there should not be any error in terms of the data types and declarations.

6 Test Plan

6.1 Testing phases

6.1.1 Unit Test

We adopted a Test-Driven Model in DAVE's development. Every feature comes in a company with a corresponding test case, and it is only marked as completed when the the test case passed. A collection of expect-to-fail test cases is also included in the test suite, mostly to verify if the Semantic Analysis part of the DAVE compiler is functioning normally. As its name implies, they only pass when they are failing to compile.

6.1.2 System Test

System Test is conducted at the very end of DAVE's development, trying to eliminate any potential bug incurred by multiple components working together (and bugs that get away with the unit test). We tried to "break" the compiler as much as we could and fixed discovered bugs.

6.2 Testing suites

Our test plan are covered in three folders, the pass folder for the test cases that should pass, the ideal folder to store the intending result of these test cases, allowing us to implementing linear test, compiling and executing all the test cases at the same and and return whether exist difference. Also, we have a fail folder, checking the robustness of our semantic and syntax checking. The test files could be run under the /src folder invoking the command line of ./test.sh.

We simply categorize the functions of our testing files.
Case 1-5 : testing the definition of variable, their conversion and comparison.
Case 6-14: testing mostly on the clause statement, mainly for loop, while loop and if-else statement, also including the testing for function initialization and the construction of primitive array.

Case 15-20: testing cases for the definition of the novel data types, meaning the record, field and table. Allow several kinds of construction method mentioned in the Language Reference Manual.

Case 21-25, 32: testing the table modification, table appending with records, fields or other tables. These seems to be the most fundamental among all the build-up functions. Also we have the converting of fld attribute type here.

Case 26-29: testing the table access, means retrieving field or record from the current table, but because of some object causes, part of it (case 28 and 29) still need further examination and complementation.

Case 30-31: testing the file output and input of DAVE language, examining whether the input and output file are consistent, to better ameliorate the formats. At the same time, it is the basis of our further development on data manipulation and utilization.

Case 34-37: testing some arithmetic operation of table, like the add, minus, multiplication and division of, since it would be helper under several situations, only the numerable part would be considered.

The test cases begun from the Case 40 include those for the bad cases.
Case 40-43: testing the lexical and syntactic problems, to make sure the parser could correctly translate in the related AST.
Case 46-50: testing the definition of array, table, record and field. Because the definition is slightly different from the C syntax, so most of the cases would concentrate on it.
Case 51-55: Other testing files for the statements, like the examination of function definition and the loop statements.

6.3 Demo Programs

Here are our two demo programs shown in the presentation, the merge-sort algorithm and the linear regression function.

6.3.1 Merge-Sort Algorithm

- **DAVE Code**

```
int main() {
  int[] A=[6,9,3,10];
  int[] B=[0,0,0,0];
  print("Before");
  print(A[0]);
  print(A[1]);
  print(A[2]);
```

```
  print(A[3]);
  mergeSort(A,B,4);
  print("After");
  print(A[0]);
  print(A[1]);
  print(A[2]);
  print(A[3]);
}

void mergeSort(int[] A, int[] B, int n) {
  int w = 1;
  int i = 0;
  for (w = 1; w < n; w = 2 * w) {
     for (i = 0; i < n; i = i + 2 * w) {
        bupMerge(A, i, min(i+w, n), min(i+2*w, n), B);
     }
     cpyArr(B,A,n);
   }
}

void bupMerge(int[] A, int left, int right, int end, int[] B) {
   int i = left;
   int j = right;
   int k = 0;
   for (k = left; k < end; k++) {
      if (i < right && (j >= end || A[i] <= A[j])) {
         B[k] = A[i];
         i = i + 1;
      } else {
         B[k] = A[j];
         j = j + 1;
      }
   }
}

void cpyArr (int[] B, int[] A, int n) {
  int i = 0;
   for (i = 0; i < n; i++) {
      A[i] = B[i];
   }
}

int min(int a, int b) {
```

```
 if (a>=b) {
  return b;
 } else {
  return a;
 }
}
```

- **Generated C++ Code**

```
#include "dave.h"

int max(fld a) {
{
int max;
int current = get_int(a, 0);
int length = (a.length);
int i;
for ((i = 1); (i < length); (i++)) {
if ((get_int(a, i) < current))
{
((max = current));
}

}

return max;
}

}
int min(fld a) {
{
int min;
int i;
int length = (a.length);
int current = get_int(a, 0);
for ((i = 1); (i < length); (i++)) {
if ((get_int(a, i) > current))
{
((min = current));
}

}
return min;
}
```

```
}

int min(int a, int b) {
{
if ((a >= b))
{
return b;
}
else {
return a;
}
}

}
void cpyArr(vector<int> B, vector<int> A, int n) {
{
int i = 0;
for ((i = 0); (i < n); (i++)) {
(((A[i]) = (B[i])));
}

}

}
void bupMerge(vector<int> A, int left, int right, int end, vector<int> B) {
{
int i = left;
int j = right;
int k = 0;
for ((k = left); (k < end); (k++)) {
if (((i < right) && ((j >= end) || ((A[i]) <= (A[j])))))
{
(((B[k]) = (A[i])));
((i = (i + 1)));
}
else {
(((B[k]) = (A[j])));
((j = (j + 1)));
}

}
```

```
}

}
void mergeSort(vector<int> A, vector<int> B, int n) {
{
int w = 1;
int i = 0;
for ((w = 1); (w < n); (w = (2 * w))) {
for ((i = 0); (i < n); (i = (i + (2 * w)))) {
(bupMerge(A, i, min((i + w), n), min((i + (2 * w)), n), B));
}

(cpyArr(B, A, n));
}

}
}
int main() {
{
int _A[] = {6, 9, 3, 10};
vector<int> A = to_vector(_A, getArrayLen(_A));
int _B[] = {0, 0, 0, 0};
vector<int> B = to_vector(_B, getArrayLen(_B));
(print("Before"));
(print((A[0])));
(print((A[1])));
(print((A[2])));
(print((A[3])));
(mergeSort(A, B, 4));
(print("After"));
(print((A[0])));
(print((A[1])));
(print((A[2])));
(print((A[3])));
}

}
```

6.3.2 Simple Linear Regression (Table-Based)

- **DAVE Code**

```
int main() {
```

```
      float[] x=[1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0];
      float[] y=[7.0,2.0,9.0,0.0,5.0,4.0,10.0,900.0,1.0,23.0];
      fld f1 = new fld(x,"ind");
      fld f2 = new fld(y,"dep");
      str a = "ind";
      str b = "dep";
      fld[] f = [f1,f2];
      tbl sample = new tbl(f);
      print("Original Dataset:");
      print(sample);
      tbl newsample = simpleLinearRegression(sample,a,b);
      print("Updated Dataset");
      print(newsample);

}
tbl simpleLinearRegression(tbl sample, str ind, str dep) {
  int sizeX = sample.col_length;
  float avgX = 0.0;
  float avgY = 0.0;
  float sumX = 0.0;
  float sumY = 0.0;
  float cov = 0.0;
  float var = 0.0;
  int i = 0;
  fld x = access(sample,ind);
  fld y = access(sample,dep);
  for (i = 0; i < sizeX; i++) {
    sumX = sumX + get_float(x, i);
    sumY = sumY + get_float(y, i);
  }
  avgX = sumX/sizeX;
  avgY = sumY/sizeX;
  for (i = 0; i < sizeX; i++) {
    cov = cov + ((get_float(x, i)-avgX)*(get_float(y, i)-avgY));
    var = var + ((get_float(x, i)-avgX)*(get_float(x, i)-avgX));
  }
  float beta = cov/var;
  float alpha = avgY - beta * avgX;
  float[sizeX] f;
  for (i = 0; i < sizeX; i++) {
    f[i] = alpha + beta * get_float(x, i);
  }
  fld f3 = new fld(f,"est");
```

```
  float ssres = 0.0;
  float sstot = 0.0;
  for (i = 0; i < sizeX; i++) {
    ssres = ssres + ((get_float(y, i)-f[i])*(get_float(y, i)-f[i]));
    sstot = sstot + ((get_float(y, i)-avgY)*(get_float(y, i)-avgY));
  }
  float rsquare = 1 - ssres/sstot;
  print("The beta value is");
  print(beta);
  print("The alpha value is");
  print(alpha);
  print("The R-Square value is (%)");
  print(rsquare*100);
  tbl newsample = append(sample,f3);
  return newsample;
}
```

- **Generated C++ Code**

```
#include "dave.h"

int max_value(fld a) {
{
int max;
int current = get_int(a, 0);
int length = (a.length);
int i;
for ((i = 1); (i < length); (i++)) {
if ((get_int(a, i) < current))
{
((max = current));
}

}

return max;
}

}
int min_value(fld a) {
{
int min;
int i;
int length = (a.length);
```

```
int current = get_int(a, 0);
for ((i = 1); (i < length); (i++)) {
if ((get_int(a, i) > current))
{
((min = current));
}

}

return min;
}

}

tbl simpleLinearRegression(tbl sample, string ind, string dep) {
{
int sizeX = (sample.col_length);
double avgX = 0.;
double avgY = 0.;
double sumX = 0.;
double sumY = 0.;
double cov = 0.;
double var = 0.;
int i = 0;
fld x = access(sample, ind);
fld y = access(sample, dep);
for ((i = 0); (i < sizeX); (i++)) {
((sumX = (sumX + get_float(x, i))));
((sumY = (sumY + get_float(y, i))));
}

((avgX = (sumX / sizeX)));
((avgY = (sumY / sizeX)));
for ((i = 0); (i < sizeX); (i++)) {
((cov = (cov + ((get_float(x, i) - avgX) * (get_float(y, i) -
avgY)))));
((var = (var + ((get_float(x, i) - avgX) * (get_float(x, i) -
avgX)))));
}

double beta = (cov / var);
double alpha = (avgY - (beta * avgX));
vector<double> f(sizeX);
```

```
for ((i = 0); (i < sizeX); (i++)) {
(((f[i]) = (alpha + (beta * get_float(x, i)))));
}

fld f3 = fld(&f[0], "est", f.size());
double ssres = 0.;
double sstot = 0.;
for ((i = 0); (i < sizeX); (i++)) {
((ssres = (ssres + ((get_float(y, i) - (f[i])) * (get_float(y, i) -
(f[i])))));
((sstot = (sstot + ((get_float(y, i) - avgY) * (get_float(y, i) -
avgY)))));
}

double rsquare = (1 - (ssres / sstot));
(print("The beta value is"));
(print(beta));
(print("The alpha value is"));
(print(alpha));
(print("The R-Square value is (%)"));
(print((rsquare * 100)));
tbl newsample = append(sample, f3);
return newsample;
}

}
int main() {
{
double _x[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
vector<double> x = to_vector(_x, getArrayLen(_x));
double _y[] = {7., 2., 9., 0., 5., 4., 10., 900., 1., 23.};
vector<double> y = to_vector(_y, getArrayLen(_y));
fld f1 = fld(&x[0], "ind", x.size());
fld f2 = fld(&y[0], "dep", y.size());
string a = "ind";
string b = "dep";
fld _f[] = {f1, f2};
vector<fld> f = to_vector(_f, getArrayLen(_f));
tbl sample = tbl(&f[0], f.size(), f[0].length);
(print("Original Dataset:"));
(print(sample));
tbl newsample = simpleLinearRegression(sample, a, b);
(print("Updated Dataset"));
```

```
(print(newsample));
}

}
```

6.3.3 Simple Linear Regression (Array-Based)

Another similar simple linear regression program is added under /showcase folder. The program functions exactly the same with the showcase program in Section 6.1.2, except for the fact that it utilizes the array type instead of tbl, fld and rec type. The code is not presented in this report to avoid the inclusion of large blocks of duplicate code. Check /showcase/linearRegression (Array-Based).dave and /showcase/linearRegression (Array-Based, Sample) to find them.

7 Lesson Learned

7.1 Fan Yang

As the Professor Edwards implied in the first class, the Programming Language and Translator course do teach me a lot. Not only in the technical related knowledge about compiler, but also the skills to organize a group project and realize it. Planning and communication are two most crucial factors in carrying on a project. Without planning, the endeavor of teammates would seems in vain as we could not dedicate on the right direction. An appropriate division of time would help us grasp the initiative of completing the project, but hurried to do so while the deadline is imminent. Also, communication is worthwhile, the front end and the back end of the compiler need to be coordinated, the target language library and the DAVE language should also be coordinated. So without enough communication between each part of the project, it could not be realized smoothly. At last, though most of my works concentrate on the C++ library and the test, I have also learnt such valuable experience on Ocaml programming and compiler implementation. Thanks again for Prof. Edwards, my three teammates, especially James and the TA in charge of our DAVE project, Prachi.

7.2 James HyunSeung Hong

I was the manager of the team, and it was sometimes difficult to coordinate the weekly meeting. However, once we have finalized the weekly meeting time, we tried our best to be committed to the set schedule. Although it was very difficult to learn how to use Ocaml at first, once I got the grasp of it I realized that it is a very efficient and powerful language. Due to time constraints and syntax changes throughout the semester, we couldn't implement all the functionalities that we wanted to have in the first place. For example, we couldn't implement the visualization part. Instead, we decided to focus more on the data analysis part by implementing table multiplIcation, addition, subtraction, field and record manipulations and saving/loading data from a text file. If we had more time to work on it and add more features, we would certainly have added the visualization functionalities. Overall, it was a very worthwhile experience as I was able to learn how programming languages that we use everyday actually work.

7.3 Min Woo Kim

First of all, I was humbled to realize that there still exist many powerful languages that I am not acquainted with. I was particularly impressed with Ocaml's succinct syntax and pattern matching functionalities. I very much enjoyed the entire development process of designing, building, debugging, and preparing the manual. By developing a mini language on my own, I could finally understand how computer programming languages actually operate and how they are architectured.

Moreover, It was the most time consuming and demanding group project I ever did at Columbia, and I was able to experience the team dynamics and project management while working closely with my peers. I learned from my teammates as much as I learned in class. Overall, it was a very revealing learning experience for me, both technically and non-technically, and I am glad that I have successfully completed this course.

7.4 Chen Yu

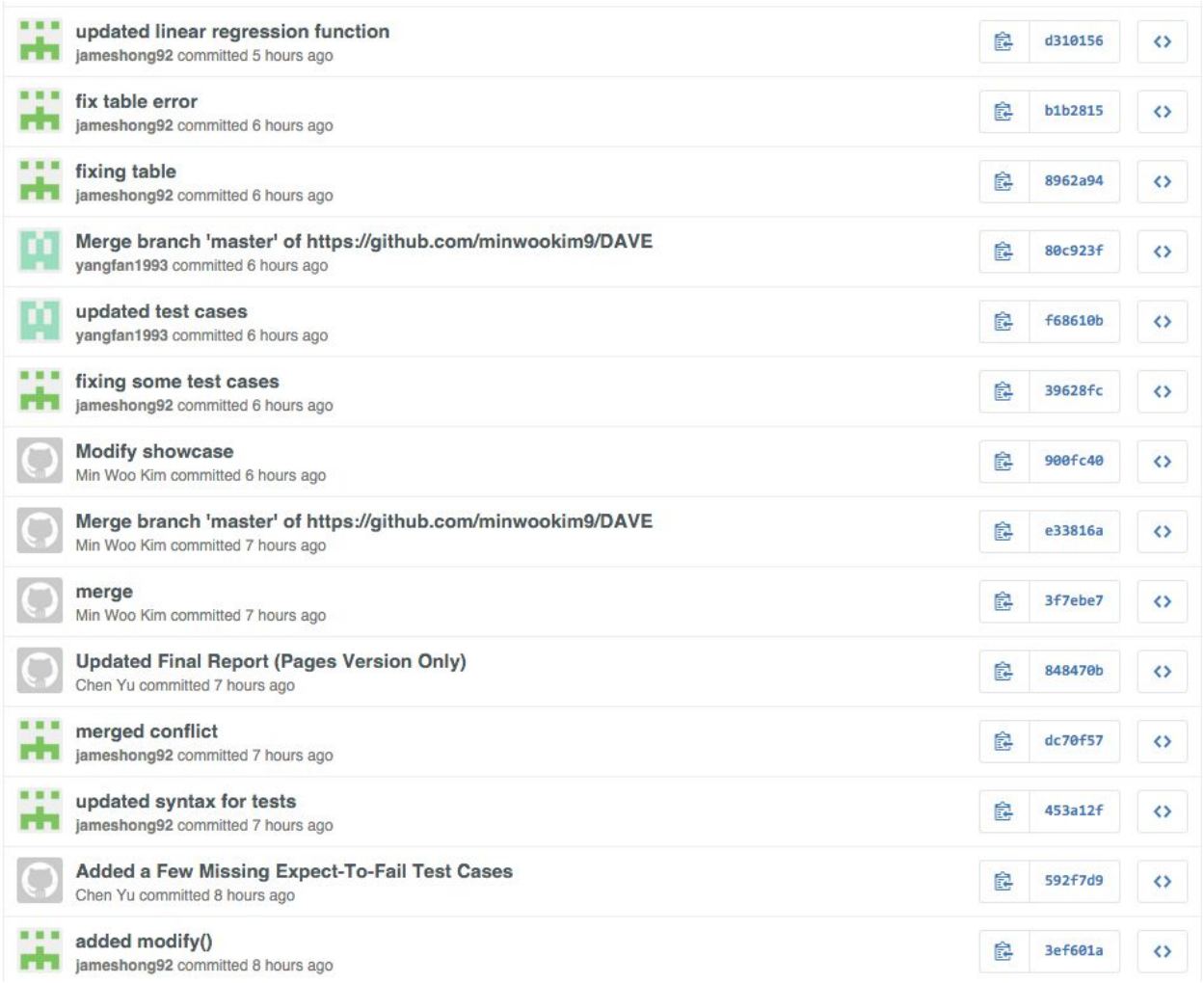
Trying to create a programming language of our own is an eye-opening experience, not only because of all the new ideas and theories involved in the process, but also for the fact that it grants us an unique opportunity to reexamine a lot of tools we take for granted in everyday software development. With today's convenient integrated development environments, we tend to forget about all the delicacies under the hood, and building something from bottom to the top does have the magic to make people recollect these hidden little wonders.

DAVE is by no means easy to build. All of us have other tasks to worry about throughout the semester and it is really difficult to coordinate efforts. Worst of all, contingencies happen, and we are forced to figure out a way to fill things up. Had we granted another chance, we might be able to plan things better ahead and finish more than we are doing right now. However, it is exactly from those kinds of experiences that we begin to learn and grow, for which we are deeply thankful for.

Appendix A

Source Code Management

We used GitHub for our primary tool for distributed revision control and code management. The snapshot of the commit history (333 commits total) is provided below:



Appendix B

Built-in Functions and C++ Library

Except the functions explicitly outlined in the Reference Manual, the DAVE language still have several useful and negligible built in functions in the C++ library. Thus assists the amateurs to easy get in touch with the language.

These build in functions includes:

- **Append Function**

The append functions could have wide utilization while combining two tables, or binding a new table or record to the existing table. Without it, the flexible operation of table is hard to realize.
The function have the including two invoke methods.

`tbl e = append(tbl, fld);` //combine a fld to the right of the table. If the name has been used, it would not be permitted.

`tbl e = append(tbl, rec);` //combine a rec to the end of the table. Here the sequence of the rec could be inconsistent with the tbl also.

`tbl e = append(tbl, tbl, bool method);` // if method equals 0, the combination of two table are horizontally, means in the left, right; if method equals 1, they are combined vertically, in the up, down direction.

- **Access Function**

The access functions help us grasp record and field from the existing table, thus accelerate our in fetching crucial information from the table. It could also be used as an way to construct novel record and field from the out-dated date.

`rec g = access(tbl, int num);` // to get the corresponding record of the table, the record in marked by the row number.

`fld h = access(tbl, string name);` // to get the corresponding field of the table, unlike the rec accessing, the field is marked by its own name.

- **Convert Function**

Also in the DAVE language, the construction of records, fields and tables do not need to explicitly define the primitive type, the compiler would find out it automatically. However, when we read from the file, the data would be implicitly set as str. If we need

further arithmetic operation, or more interesting like linear regression, changing the data type is critical, so we built in the convert function here.

```
tbl f = convert(tbl, col num, type);
```

the type is a string, it could be "int", "double", or "string". As for “bool”, we seldom use it, so the conversion need manually. It is to convent the corresponding column to the new data type. Also remember here, the num are counted from zero but not one.

- **Modify Function**

Punctuality is also a significant element of the maintaining or dataset. So the table would need to be updated routinely so as to keep the correctness of it, and without emphasizing the remaining other elements, so a modify function is indispensable. The example is like the following:

```
tbl f = modify(tbl, row num, col num, data);
```

data could be int, double or string, change the value in the (col pos, row pos) of the original table to a new value. To be reminded, here the position of column and row are started from 0 insted of 1, so During the daily maintenance, both the position typically need to be reduced by 1. Also, The data type should be the same as the data type of the inserting fld, if not, the compiler would not know what to fetch.

Appendix C

Built-in Dave Library

The standard DAVE package includes a built-in library written in DAVE language, named “dave_core.dave.” You can refer to the source file included in Appendix D. The library includes functions for calculating *max_value*, *min_value*, and *mean* values of fld type variables. They return float values.

Appendix D

Source Files

ast.ml

```
(*ast.ml*)
exception Invalid_type of string
exception Syntax_error of string

(*Binary Operators (In Order): +, -, *, /, %, ^, ==, !=, <, >, >=, <=, &&, ||*)
```

```
type binop = Add | Sub | Mul | Div | Mod | Exp | Eq | Neq | Lt | Gt | Leq | Geq | And | Or
(*Unary Operators (Before Operand) (In Order): !, -*)
type unop = Not | Neg
(*Unary Operators (After Operand) (In Order): ++, --*)
type postop = Inc | Dec
(*Assignment Operators = += -= *= /= %=*)
type asnop = Asn | Addeq | Subeq | Muleq | Diveq | Modeq
(*Supported Datatypes*)
type datatype = Int | Float | String | Bool | Fld | Tbl | Rec | Void
(*Arguments*)
type arg = datatype * string

type var = {
        ptype: datatype;
        dimension: expr list;
}
and expr =
        Var of string
        | Array of string * expr
        | Access of expr * string
        | IntLit of int
        | FloatLit of float
        | StringLit of string
        | BoolLit of bool
        | ArrayLit of expr list
        | Range of expr * expr
        | Binop of expr * binop * expr
        | Unop of unop * expr
        | Postop of expr * postop
        | AssignOp of expr * asnop * expr
        | Lval of expr
        | Cast of var * expr
        | FuncCall of string * expr list
        (*Tbl = a list of Rec | a list of Fld*)
        | Tbl of expr list
        (*Rec = list of id (key) * Value of Each Component*)
        (*Supplemental: Store Values of Each Component in the Format of String When Coding the Parser*)
        | Rec of expr list
        | RecRef of string * expr
        (*Fld = Values of the List * Name of the Field*)
        | Fld of expr * string
        | Noexpr
        | None

type decl = {
        vname: string;
        vtype: var;
        vinit: expr;
}

(*Statement*)
type stmt =
        Expr of expr
        | Return of expr
        | Block of stmt list
        | If of expr * stmt * stmt
        | For of expr * expr * expr * stmt
```

```
        | While of expr * stmt
        | VarDeclStmt of decl
        | Continue
        | Break
        | EmptyStmt

(*Function Declaration*)
type func_decl = {
        fname : string;
        formals : decl list;
        body: stmt;
        return_type : var;
}

(*Programs*)
type program = {
        gdecls : decl list;
        fdecls : func_decl list;
}


(*Printing AST*)
let string_of_binop = function
  Add -> "+"
| Sub -> "-"
| Mul -> "*"
| Div -> "/"
| Mod -> "%"
| Exp -> "^"
| Eq -> "=="
| Neq -> "!="
| Lt -> "<"
| Gt -> ">"
| Leq -> "<="
| Geq -> "<="
| And -> "&&"
| Or -> "||"

let string_of_asnop = function
  Asn -> "="
| Addeq -> "+="
| Subeq -> "-="
| Muleq -> "*="
| Diveq -> "/="
| Modeq -> "%="

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

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

let rec string_of_datatype = function
  Int -> "int"
| Float -> "float"
```

```
| Bool -> "bool"
| String -> "str"
| Tbl -> "tbl"
| Rec -> "rec"
| Fld -> "fld"
| Void -> "void"

let rec string_of_expr = function
  IntLit(lit) -> "IntLit( " ^ string_of_int lit ^ " )"
| FloatLit(lit) -> "FloatLit( " ^ string_of_float lit ^ " )"
| StringLit(lit) -> "StringLit( " ^ lit ^ " )"
| BoolLit(lit) -> "BoolLit( " ^ string_of_bool lit ^ " )"
| ArrayLit(exps) -> "ArrayLit( " ^ String.concat ", " (List.map string_of_expr exps) ^ " )"
| Range(exp1, exp2) -> "Range( " ^ string_of_expr exp1 ^ " " ^ string_of_expr exp2 ^ " )"
| Binop(exp1, binop, exp2) -> "Binop( " ^ string_of_expr exp1 ^ " " ^ string_of_binop binop ^ " " ^
string_of_expr exp2 ^ " )"
| Unop(unop, exp) -> "Unop( " ^ string_of_unop unop ^ " " ^ string_of_expr exp ^ " )"
| Postop(lvalue, postop) -> "Postop( " ^ string_of_expr lvalue ^ " " ^ string_of_postop postop ^ " )"
| AssignOp(lvalue, asnop, exp) -> "AssignOp( " ^ string_of_expr lvalue ^ " " ^ string_of_asnop asnop ^
" " ^ string_of_expr exp ^ " )"
| Lval(lvalue) -> "Lval( " ^ string_of_expr lvalue ^ " )"
| Cast(var, exp) -> "Cast( " ^ string_of_vtype var ^ " " ^ string_of_expr exp ^ " )"
| FuncCall(id, exps) -> "FuncCall( " ^ id ^ ", " ^ String.concat "; " (List.map string_of_expr exps) ^
" )"
| Tbl(exps) -> "Tbl( " ^ String.concat "; " (List.map string_of_expr exps) ^ " )"
| Rec(exps) -> "Rec{ " ^ String.concat ", " (List.map string_of_expr exps) ^ " }"
| RecRef(id, exp) -> "RecRef( " ^ id ^ ":" ^ string_of_expr exp ^ " )"
| Fld(exp, lit) -> "Fld(" ^ string_of_expr exp ^ ", '" ^ lit ^ "' )"
| Noexpr -> "Noexpr"
| None -> "None"
| Var(lit) -> "Var( " ^ lit ^ " )"
| Array(id, index) -> "Array( " ^ id ^ "[" ^ string_of_expr index ^ "] ) "
| Access(exp, id) -> "Access( " ^ string_of_expr exp ^ " " ^ id ^ " )"

and string_of_vtype v =
let dimension = v.dimension in
        match dimension with
        [] -> string_of_datatype v.ptype
        | _ -> string_of_datatype v.ptype ^ "[" ^ string_of_int (List.length v.dimension) ^ "]"

let string_of_decl vdecl =
        let init = vdecl.vinit in
        match init with
        Noexpr ->
                ("Var( " ^ string_of_vtype vdecl.vtype ^ " " ^ vdecl.vname ^ " )")
        | _ ->
                ("Var( " ^ string_of_vtype vdecl.vtype ^ " " ^ vdecl.vname ^ " = " ^ string_of_expr
vdecl.vinit ^ " )")

let rec string_of_stmt = function
  Expr(exp) -> "Expr( " ^ string_of_expr exp ^ " )"
| Return(exp) -> "Return( " ^ string_of_expr exp ^ ")"
| Block(stmt_list) -> (String.concat "\n" (List.map string_of_stmt stmt_list)) ^ "\n"
| If(exp, stmt1, stmt2) -> "if (" ^ string_of_expr exp ^ ") {\n" ^ string_of_stmt stmt1 ^
string_of_stmt stmt2 ^ "\n}"
| For(init, test, after, stmt) -> "for (" ^ string_of_expr init ^ ", " ^ string_of_expr test ^ ", " ^
string_of_expr after ^ ") {\n" ^ string_of_stmt stmt ^ "\n}"
```

```
| While(test, stmt) -> "while( " ^ string_of_expr test ^ " ) {\n" ^ string_of_stmt stmt ^ "\n}"
| VarDeclStmt(var) -> "VarDeclStmt( " ^ (string_of_decl var) ^ " )"
| Continue -> "Continue;\n"
| Break -> "Break\n"
| EmptyStmt -> "EmptyStmt\n"

let string_of_func_decl funcdecl = "Function( return type: ("
        ^ string_of_vtype funcdecl.return_type ^ ") name: \""
        ^  funcdecl.fname ^ "\" formals: ("
  ^ (String.concat ", " (List.map string_of_decl funcdecl.formals))
  ^ ") {\n"
        ^ string_of_stmt funcdecl.body ^ "\n}"

let string_of_program prgm = "Program( "
        ^ (String.concat "\n" (List.map string_of_decl prgm.gdecls)) ^ "\n\n"
        ^ (String.concat "\n\n" (List.map string_of_func_decl prgm.fdecls)) ^ "\n)\n"

let type_of_string = function
        "int" -> Int
        | "float" -> Float
        | "bool" -> Bool
        | "str" -> String
        | "fld" -> Fld
        | "rec" -> Rec
        | "tbl" -> Tbl
        | "void" -> Void
        | dtype -> raise (Invalid_type dtype)
```

scanner.mli

```
{
        open Parser
}

let digit = ['0' - '9']
let letter = ['a'-'z' 'A'-'Z']

rule token = parse
[' ' '\t' '\r' '\n']                                    { token lexbuf }
| "/*"                                                  { comment lexbuf }
| "//"                                                  { singlelinecomment lexbuf }
| '('                                                   { LPAREN }
| ')'                                                   { RPAREN }
| '['                                                   { LBRACK }
| ']'                                                   { RBRACK }
| '{'                                                   { LBRACE }
| '}'                                                   { RBRACE }
| ';'                                                   { SEMICOL }
| ':'                                                   { COLON }
| ','                                                   { COMMA }
| '.'                                                   { DOT }
| '+'                                                   { PLUS }
| '-'                                                   { MINUS }
| '*'                                                   { TIMES }
| '/'                                                   { DIVIDE }
```

```
| '%'                                                                 { MOD }
| '^'                                                                 { EXP }
| "+="                                                                { ADDEQ }
| "-="                                                                { SUBEQ }
| "*="                                                                { MULEQ }
| "/="                                                                { DIVEQ }
| "%="                                                                { MODEQ }
| "++"                                                                { INC }
| "--"                                                                { DEC }
| '='                                                                 { ASN }
| "&&"                                                                { AND }
| "||"                                                                { OR }
| '!'                                                                 { NOT }
| "=="                                                                { EQ }
| "!="                                                                { NEQ }
| '<'                                                                 { LT }
| "<="                                                                { LEQ }
| ">"                                                                 { GT }
| ">="                                                                { GEQ }
| "if"                                                                { IF }
| "else"                                                              { ELSE }
| "for"                                                                      { FOR }
| "while"                                                             { WHILE }
| "break"                                                             { BREAK }
| "continue"                                                          { CONTINUE }
| "return"                                                            { RETURN }
| "void"                                                              { VOID }
| "int"                                                                      { INT }
| "float"                                                             { FLOAT }
| "bool"                                                              { BOOL }
| "str"                                                                      { STR }
| "tbl"                                                                      { TBL }
| "rec"                                                                      { REC }
| "fld"                                                                      { FLD }
| "none"                                                                     { NONE }
| "new"                                                               { NEW }
| "true"                                                              { BOOL_LIT(true) }
| "false"                                                             { BOOL_LIT(false) }
| (digit)+ as lit                                                     { INT_LIT(int_of_string lit) }
| ((digit)*'.'(digit)+ | (digit)+'.') as lit
                                                                      { FLOAT_LIT(float_of_string lit) }
| '"' ([^ '"' '\\' '\n' '\r' '\t']* ('\\' ['\\' '"' 'n' 'r' 't'])* as lit) '"'
                                                                      { STR_LIT(lit) }
| letter (letter | digit | '_')* as lit
                                                                      { ID(lit) }
| eof                                                                 { EOF }
| _ as c
       { raise (Failure("Found illegal character: " ^ Char.escaped c)) }

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


and singlelinecomment = parse
"\n"            { token lexbuf }
| eof           { EOF }
```

```
| _            { singlelinecomment lexbuf }
```

parser.mly

```
%{ open Ast %}
%token NEW
%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token SEMICOL COMMA DOT COLON
%token ADDEQ SUBEQ MULEQ DIVEQ MODEQ
%token PLUS MINUS TIMES DIVIDE MOD INC DEC EXP
%token ASN AND OR NOT
%token EQ NEQ LT LEQ GT GEQ
%token CONTINUE BREAK IF ELSE FOR WHILE RETURN
%token FLD TBL REC
%token VOID STR BOOL INT FLOAT FLD TBL REC
%token <int> INT_LIT
%token <float> FLOAT_LIT
%token <bool> BOOL_LIT
%token <string> STR_LIT ID
%token NONE
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASN INC DEC ADDEQ SUBEQ MULEQ DIVEQ MODEQ
%right DOT
%left LBRACK
%left OR
%left AND
%left EQ NEQ
%left LT LEQ GT GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD EXP
%right NOT UNMINUS

%start program
%type <Ast.program> program

%%

/* Expressions */

/* str int float bool rec fld tbl str[] int[] float[] bool[] fld[] rec[] */
datatype:
        INT
{

                        {

                                ptype = Int;

                                dimension = []

                        }
```

```
                }
| STR {
                        {
                                ptype = String;
                                dimension = []
                        }
                }
| FLOAT {
                        {
                                ptype = Float;
                                dimension = []
                        }
                }
| BOOL {
                        {
                                ptype = Bool;
                                dimension = []
                        }
                }
| FLD {
                        {
                                ptype = Fld;
                                dimension = []
                        }
                }
| REC {
                        {
                                ptype = Rec;
                                dimension = []
                        }
                }
```

```
| TBL {

                        {

                                ptype = Tbl;

                                dimension = []

                        }

                }
| VOID
                {

                        {

                                ptype = Void;

                                dimension = []

                        }

                }
| datatype LBRACK expr_opt RBRACK

                {

                        {

                                ptype = $1.ptype;

                                dimension = [(if $3 == Noexpr then IntLit(0) else $3)]

                        }

                }

/* id[index_list], expr.id */
lvalue:
        ID
        { Var($1) }
        | ID LBRACK index_list RBRACK

                { Array($1, $3) }

index_list:
        expr                                                    { $1 }
        | COLON expr                                    { Range(Noexpr, $2) }
        | expr COLON                                    { Range($1, Noexpr) }
        | expr COLON expr                               { Range($1, $3) }

expr:
        | expr PLUS expr                                        { Binop($1, Add, $3) }
        | expr MINUS expr                               { Binop($1, Sub, $3) }
        | expr TIMES expr                               { Binop($1, Mul, $3) }
        | expr DIVIDE expr                              { Binop($1, Div, $3) }
        | expr MOD expr                                         { Binop($1, Mod, $3) }
```

```
        | expr EXP expr                                   { Binop($1, Exp, $3) }
        | expr EQ expr                                  { Binop($1, Eq, $3) }
        | expr NEQ expr                                 { Binop($1, Neq, $3) }
        | expr LT expr                                  { Binop($1, Lt, $3) }
        | expr GT expr                                  { Binop($1, Gt, $3) }
        | expr LEQ expr                                 { Binop($1, Leq, $3) }
        | expr GEQ expr                                 { Binop($1, Geq, $3) }
        | expr AND expr                                 { Binop($1, And, $3) }
        | expr OR expr                                  { Binop($1, Or, $3) }

        | NOT expr                                      { Unop(Not, $2) }
        | MINUS expr %prec UNMINUS                      { Unop(Neg, $2) }
        | NONE                                          { None }

        | lvalue INC                                    { Postop($1, Inc) }
        | lvalue DEC                                    { Postop($1, Dec) }

        | lvalue ADDEQ expr                             { AssignOp($1, Addeq, $3) }
        | lvalue SUBEQ expr                             { AssignOp($1, Subeq, $3) }
        | lvalue MULEQ expr                             { AssignOp($1, Muleq, $3) }
        | lvalue DIVEQ expr                             { AssignOp($1, Diveq, $3) }
        | lvalue MODEQ expr                             { AssignOp($1, Modeq, $3) }
        | lvalue ASN expr                               { AssignOp($1, Asn, $3) }
        | expr DOT ID                                   { Access($1, $3) }
        | lvalue                                        { Lval($1) }

        | LPAREN expr RPAREN                            { $2 }

        | datatype LPAREN expr RPAREN                   { Cast($1, $3) }
        | literal                                       { $1 }
        | tbl_lit                                       { Tbl($1) }

        | ID LPAREN actuals_opt RPAREN          { FuncCall($1, $3) }

literal:
        INT_LIT                                         { IntLit($1) }
        | FLOAT_LIT                                     { FloatLit($1) }
        | STR_LIT                                       { StringLit($1) }
        | BOOL_LIT                                      { BoolLit($1) }
        | fld_lit                                       { $1 }
        | rec_lit                                       { $1 }
        | LBRACK actuals_list RBRACK                    { ArrayLit(List.rev $2) }

tbl_lit:
        NEW TBL LPAREN actuals_list RPAREN
                                                        { List.rev $4 }

rec_lit:
        NEW REC LPAREN rec_init RPAREN
                                                        { Rec(List.rev $4) }

rec_init:
        ID COLON literal                                { [RecRef($1, $3)] }
        | rec_init COMMA ID COLON literal
                                                        { RecRef($3, $5) :: $1 }

fld_lit:
```

```
        NEW FLD LPAREN expr COMMA STR_LIT RPAREN
                                                                { Fld($4, $6) }

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

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

/* Declarations */
program:
  /* nothing */                                           { { gdecls = []; fdecls = [] } }
  | program vdecl       { { gdecls = $2 :: $1.gdecls; fdecls = $1.fdecls } }
  | program fdecl       { { gdecls = $1.gdecls; fdecls = $2 :: $1.fdecls } }

fdecl:
        datatype ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE

                {

                        {

                                fname = $2;

                                formals = $4;

                                body = Block(List.rev $7);

                                return_type = $1

                        }

                }

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

formal_list:
        datatype ID {[

                        {

                                vname = $2;

                                vtype = $1;

                                vinit = Noexpr;

                        }]

                        }
        | formal_list COMMA datatype ID

                { ({
```

```
                                vname = $4;

                                vtype = $3;

                                vinit = Noexpr;

                        }) :: $1

                }

vdecl:
        datatype ID SEMICOL                             {

                        {

                                vname = $2;

                                vtype = $1;

                                vinit = Noexpr;

                        }

                }
        | datatype ID ASN expr SEMICOL

                {

                        {

                                vname = $2;

                                vtype = $1;

                                vinit = $4;

                        }

                }

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

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

stmt:
        expr SEMICOL                                    { Expr($1) }
        | RETURN expr SEMICOL                           { Return($2) }
        | LBRACE stmt_list RBRACE                       { Block(List.rev $2) }
        | IF LPAREN expr RPAREN stmt ELSE stmt

                                                                { If($3, $5, $7) }
        | IF LPAREN expr RPAREN stmt %prec NOELSE
```

```
                                                        { If($3, $5, EmptyStmt) }
        | FOR LPAREN expr_opt SEMICOL expr_opt SEMICOL expr_opt RPAREN stmt
                                                        { For($3, $5, $7, $9) }
        | WHILE LPAREN expr RPAREN stmt
                                                        { While($3, $5) }
        | CONTINUE SEMICOL                          { Continue }
        | BREAK SEMICOL                             { Break }
        | vdecl                                         { VarDeclStmt($1) }
```

sast.ml

```
open Ast

type t_expr = {
  exp: expr;
  typ: s_var
}

and s_var = {
  s_ptype: datatype;
  s_dimension: t_expr list;
}

type s_decl = {
  s_vname: string;
  s_vtype: s_var;
  s_vinit: t_expr;
}

type s_stmt =
    S_Expr of t_expr
  | S_Return of t_expr
  | S_Block of s_stmt list
  | S_If of t_expr * s_stmt * s_stmt
  | S_For of t_expr * t_expr * t_expr * s_stmt
  | S_While of t_expr * s_stmt
  | S_VarDeclStmt of s_decl
  | S_Continue
  | S_Break
  | S_EmptyStmt

type s_func_decl = {
  s_fname : string;
  s_formals : s_decl list;
  s_body: s_stmt;
  s_return_type : s_var;
}

type s_program = {
  s_gdecls : s_decl list;
  s_fdecls : s_func_decl list;
}
```

semanticAnalysis.ml

```
open Ast
open Sast
open SemanticExceptions
open Printf

module StringMap = Map.Make(String)

let rec type_of_var id v_context =
        if StringMap.mem id v_context then
                fst (StringMap.find id v_context)
        else
                raise (No_variable_err id)

let rec type_of_expr f_context v_context exp = match exp with
        Var(id) -> type_of_var id v_context
        | Array(id, exp) -> type_of_array id exp f_context v_context
        | Access(exp, id) -> type_of_access exp id f_context v_context
        | IntLit(lit) -> { s_ptype = Int; s_dimension = [] }
        | FloatLit(lit) -> { s_ptype = Float; s_dimension = [] }
        | StringLit(lit) -> { s_ptype = String; s_dimension = [] }
        | BoolLit(lit) -> { s_ptype = Bool; s_dimension = [] }
        | ArrayLit(exprs) -> type_of_array_lit exprs f_context v_context
        | Range(exp1, exp2) ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
(match type1.s_ptype, type2.s_ptype with
Int, Int -> { s_ptype = Int; s_dimension = [{ exp = exp1; typ = type1 }] }
| Int, Void -> { s_ptype = Int; s_dimension = [{ exp = exp1; typ = type1 }] }
| Void, Int -> { s_ptype = Int; s_dimension = [{ exp = exp2; typ = type2 }] }
| _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ ", " ^ string_of_expr exp2))
)
        | Binop(exp1, binop, exp2) -> (
                match binop with
                Add ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
                (match type1.s_ptype, type2.s_ptype with
                        Int, Int -> type1
                        | Int, Float -> type2
                        | Float, Int -> type1
                        | Float, Float -> type1
                        | String, String -> type1
                        | Tbl, Tbl -> type1
                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ ", " ^
string_of_expr exp2))
                )
        | Eq | Neq | Lt | Gt | Leq | Geq ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
                (match type1.s_ptype, type2.s_ptype with
                        Int, Int -> { s_ptype = Bool; s_dimension = [] }
                        | Int, Float -> { s_ptype = Bool; s_dimension = [] }
```

```
                        | Float, Int -> { s_ptype = Bool; s_dimension = [] }
                        | Float, Float -> { s_ptype = Bool; s_dimension = [] }
                        | String, String -> { s_ptype = Bool; s_dimension = [] }
                        | Bool, Bool -> { s_ptype = Bool; s_dimension = [] }
                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ " " ^
string_of_datatype type1.s_ptype ^ ", " ^ string_of_expr exp2 ^ " " ^ string_of_datatype
type2.s_ptype))
                )
        | Sub | Mul | Div ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
                (match type1.s_ptype, type2.s_ptype with
                        Int, Int -> type1
                        | Int, Float -> type2
                        | Float, Int -> type1
                        | Float, Float -> type1
                        | Tbl, Tbl -> type1
                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ ", " ^
string_of_expr exp2))
                )

        | Exp | Mod ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
                (match type1.s_ptype, type2.s_ptype with
                        Int, Int -> type1
                        | Int, Float -> type2
                        | Float, Int -> type1
                        | Float, Float -> type1
                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ ", " ^
string_of_expr exp2))
                )
        | And | Or ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
                (match type1.s_ptype, type2.s_ptype with
                        | Bool, Bool -> { s_ptype = Bool; s_dimension = [] }
                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ ", " ^
string_of_expr exp2))
                )
        )
        | Unop(unop, exp) -> (
                match unop with
                Not ->
                        let type1 = type_of_expr f_context v_context exp in
                        (match type1.s_ptype with
                                Int -> type1
                                | Bool -> type1
                                | _ -> raise (Type_err ("type error for " ^ string_of_expr exp))
                        )
                | Neg ->
                        let type1 = type_of_expr f_context v_context exp in
                        (match type1.s_ptype with
                                Int -> type1
                                | Bool -> type1
                                | Float -> type1
                                | _ -> raise (Type_err ("type error for " ^ string_of_expr exp))
```

```
                        )
                )
        | Postop(exp, postop) -> (
                match postop with
                Inc | Dec ->
                        let type1 = type_of_expr f_context v_context exp in
                        (match type1.s_ptype with
                                Int -> type1
                                | _ -> raise (Type_err ("type error for " ^ string_of_expr exp))
                        )
                )
        | AssignOp(exp1, asnop, exp2) -> (
                match asnop with
                Asn ->
                        let type1 = type_of_expr f_context v_context exp1 and
                        type2 = type_of_expr f_context v_context exp2 in
                        if List.length type1.s_dimension != List.length type2.s_dimension then
                                raise (Type_err ("type error for " ^ string_of_expr exp1 ^ "type: " ^
string_of_datatype type1.s_ptype ^ ", " ^ string_of_expr exp2))
                        else if List.length type2.s_dimension != 0 && type1.s_ptype == type2.s_ptype
then
                                type2
                        else
                                (match type1.s_ptype, type2.s_ptype with
                                        Int, Int -> type2
                                        | Float, Float -> type2
                                        | String, String -> type2
                                        | Bool, Bool -> type2
                                        | Fld, Fld -> type2
                                        | Rec, Rec -> type2
                                        | Tbl, Tbl -> type2
                                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr
exp1 ^ "type: " ^ string_of_datatype type1.s_ptype ^ ", " ^ string_of_expr exp2))
                                )
        | Addeq ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
                (match type1.s_ptype, type2.s_ptype with
                        Int, Int -> type1
                        | Int, Float -> type2
                        | Float, Int -> type1
                        | Float, Float -> type1
                        | String, String -> type1
                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ ", " ^
string_of_expr exp2))
                                                        )
        | Subeq | Muleq | Diveq | Modeq ->
                let type1 = type_of_expr f_context v_context exp1 and
                type2 = type_of_expr f_context v_context exp2 in
                (match type1.s_ptype, type2.s_ptype with
                        Int, Int -> type1
                        | Int, Float -> type2
                        | Float, Int -> type1
                        | Float, Float -> type1
                        | _, _ -> raise (Type_err ("type error for " ^ string_of_expr exp1 ^ ", " ^
string_of_expr exp2))
                )
```

```
        )
| Lval(exp) -> type_of_expr f_context v_context exp
| Cast(var, exp) ->
        if var.ptype == Rec || var.ptype == Fld || var.ptype == Tbl then
                raise (Cast_err ((string_of_datatype var.ptype) ^ " needs to be initialized with
'new'"))
        else if var.ptype == Int || var.ptype == Float then
                        { s_ptype = var.ptype; s_dimension = [] }
        else raise (Cast_err "Invalid cast")
| FuncCall(fid, actuals_opt) -> type_of_func_ret fid actuals_opt f_context v_context
| Tbl(exprs) -> type_of_tbl exprs f_context v_context
| Rec(exprs) -> type_of_rec exprs f_context v_context
| RecRef(id, exp) -> type_of_rec_ref id exp f_context v_context
| Fld(expr, id) -> type_of_fld expr id f_context v_context
| Noexpr -> { s_ptype = Void; s_dimension = [] }
| None -> { s_ptype = Void; s_dimension = [] }


and type_of_tbl exprs f_context v_context =
  let expr_list = (List.map (fun expr -> type_of_expr f_context v_context expr) exprs) in
    if List.length (List.filter (fun a -> (match a.s_ptype with Fld -> true | _ -> false)) expr_list)
== (List.length exprs) ||
         List.length (List.filter (fun a -> (match a.s_ptype with Rec -> true | _ -> false)) expr_list)
== (List.length exprs) then
                { s_ptype = Tbl; s_dimension = []}
        else
                raise (Tbl_err "Table error")

and type_of_fld expr id f_context v_context =
        let first_lit = type_of_expr f_context v_context expr in
        match first_lit.s_ptype with
                Int | String | Bool | Float ->
                        if (List.length first_lit.s_dimension) > 0 then
                                        { s_ptype = Fld; s_dimension = [] }
                        else raise Fld_err
                | _ -> raise Fld_err

and type_of_rec exprs f_context v_context =
  let rec_ref_list = (List.map (fun expr -> s_check_expr f_context v_context expr) exprs) in
    if List.length (List.filter (fun a -> (match a.exp with RecRef(_,_) -> true | _ -> false))
rec_ref_list) == (List.length exprs) then
                { s_ptype = Rec; s_dimension = []}
        else
                raise Rec_err

and type_of_rec_ref id expr f_context v_context =
        let first_lit = type_of_expr f_context v_context expr in
        match first_lit.s_ptype with
                Int | String | Bool | Float ->
                        if (List.length first_lit.s_dimension) == 0 then
                                { s_ptype = first_lit.s_ptype; s_dimension = first_lit.s_dimension }
                        else raise Rec_ref_err
                | _ -> raise Rec_ref_err

and type_of_array_lit exprs f_context v_context =
        let first_lit = type_of_expr f_context v_context (List.hd exprs) in
        let check_function x =
```

```
                let p = (type_of_expr f_context v_context x) in
                (p.s_ptype = first_lit.s_ptype) && (List.length p.s_dimension) = (List.length
first_lit.s_dimension) in
                let list_filter = List.filter check_function exprs in
                        if List.length list_filter == List.length exprs then
                                let exp = {exp = IntLit(List.length exprs); typ = {s_ptype =
first_lit.s_ptype; s_dimension = []}} in
                                        { s_ptype = first_lit.s_ptype; s_dimension = [exp]}
                        else
                                raise Array_lit_err

and type_of_array id exp f_context v_context =
        if StringMap.mem id v_context then
                let type1 = fst (StringMap.find id v_context) in
                        (* check if id is of type array *)
                        if List.length type1.s_dimension >= 1 then
                                (* check if given index is of type int or range *)
                                let type2 = type_of_expr f_context v_context exp in
                                        if List.length type2.s_dimension >= 1 && type2.s_ptype == Int
then
                                                {s_ptype = type1.s_ptype; s_dimension =
type1.s_dimension}
                                        else if type2.s_ptype == Int then
                                                {s_ptype = type1.s_ptype; s_dimension = []}
                                        else
                                                raise Arr_err
                        else
                                raise Arr_err
        else
                raise Arr_err

and type_of_access exp id f_context v_context = (
        let exp_type = type_of_expr f_context v_context exp in
        match exp_type.s_ptype with
                Fld ->
                        if id = "length" then
                                {s_ptype = Int; s_dimension = []}
                        else if id = "name" then
                                {s_ptype = String; s_dimension = []}
                        else if id = "type" then
                                {s_ptype = Int; s_dimension = []}
                        else
                                raise Access_err
                | Tbl -> if id = "row_length" || id = "col_length" then {s_ptype = Int; s_dimension =
[]} else raise Access_err
                | Rec -> if id = "length" then {s_ptype = Int; s_dimension = []} else raise Access_err
                | Int | Bool | Float | String -> if (List.length exp_type.s_dimension) == 1 then
{s_ptype = Int; s_dimension = []} else raise Access_err
                | _ -> raise Access_err
)

and type_of_func_ret fid param f_context v_context =
        if StringMap.mem fid f_context then
                let s_param = List.map (fun x -> type_of_expr f_context v_context x) param in
                        let found_map = StringMap.find fid f_context in
                                try
                                        let found = List.find (match_param s_param) found_map in
```

```
                                        snd found
                                with Not_found -> raise (Invalid_func_err (fid ^ "(" ^ String.concat ",
" (List.map (fun x -> string_of_datatype x.s_ptype) s_param) ^ ")"))
        else
                raise (No_func_err fid)

and match_param p1 map =
        let p2 = fst map in
                let rec check_param t_l1 t_l2 = match t_l1, t_l2 with
                        [], [] -> true
                        | hd::tl, [] -> false
                        | [], hd::tl -> false
                        | h1::t1, h2::t2 ->
                                if h1.s_ptype == h2.s_ptype && List.length h1.s_dimension ==
List.length h2.s_dimension then
                                        check_param t1 t2
                                else
                                        false
                in check_param p1 p2

and s_check_expr f_context v_context in_exp = match in_exp with
        ArrayLit(indices) ->
                {exp = ArrayLit(List.map (fun a -> (s_check_expr f_context v_context a).exp) indices);
typ = type_of_expr f_context v_context in_exp }
        | Range(exp1, exp2) ->
                {exp = Range((s_check_expr f_context v_context exp1).exp, (s_check_expr f_context
v_context exp2).exp); typ = type_of_expr f_context v_context in_exp }
        | Binop(exp1, binop, exp2) ->
                let exp1_type = type_of_expr f_context v_context exp1 in
                        if exp1_type.s_ptype == Tbl then
                                let binop_string = match binop with Add -> "tbl_plus" | Sub ->
"tbl_minus" | Mul -> "tbl_mult" | Div -> "tbl_div" | _ -> raise (Type_err "Table arithmetic only
supported for +, -, *, /") in
                                {exp = FuncCall(binop_string, List.map (fun a -> (s_check_expr
f_context v_context a).exp) [exp1; exp2]); typ = type_of_expr f_context v_context in_exp }
                        else
                                {exp = Binop((s_check_expr f_context v_context exp1).exp, binop,
(s_check_expr f_context v_context exp2).exp); typ = type_of_expr f_context v_context in_exp }
        | Unop(unop, exp) ->
                {exp = Unop(unop, (s_check_expr f_context v_context exp).exp); typ = type_of_expr
f_context v_context in_exp }
        | Postop(lvalue, postop) ->
                {exp = Postop((s_check_expr f_context v_context lvalue).exp, postop); typ =
type_of_expr f_context v_context in_exp }
        | AssignOp(lvalue, asnop, exp2) ->
                {exp = AssignOp((s_check_expr f_context v_context lvalue).exp, asnop, (s_check_expr
f_context v_context exp2).exp); typ = type_of_expr f_context v_context in_exp }
        | Lval(lvalue) ->
                {exp = Lval((s_check_expr f_context v_context lvalue).exp); typ = type_of_expr
f_context v_context in_exp }
        | Cast(var, exp) ->
                {exp = Cast(var, (s_check_expr f_context v_context exp).exp); typ = type_of_expr
f_context v_context in_exp }
        | Array(id, indices) -> s_check_array f_context v_context in_exp id indices
        | Access(exp, id) -> s_check_access f_context v_context in_exp exp id
        | FuncCall(id, exprs) -> {exp = FuncCall(id, List.map (fun a -> (s_check_expr f_context
v_context a).exp) exprs); typ = (type_of_expr f_context v_context in_exp)}
```

```
        | Tbl(exprs) -> s_check_tbl f_context v_context in_exp exprs
        | Rec(exprs) -> s_check_rec f_context v_context in_exp exprs
        | Fld(expr, id) -> s_check_fld f_context v_context in_exp expr id
        | _ -> { exp = in_exp; typ = type_of_expr f_context v_context in_exp }

and s_check_access f_context v_context in_exp expr id =
        let exp_type = type_of_expr f_context v_context expr in (
                match exp_type.s_ptype with
                        Int | String | Bool | Float -> { exp = Access((s_check_expr f_context v_context
expr).exp, "size()"); typ = type_of_expr f_context v_context in_exp }
                        | _ -> { exp = Access((s_check_expr f_context v_context expr).exp, id); typ =
type_of_expr f_context v_context in_exp }
                )
and s_check_tbl f_context v_context in_exp exprs =
  let tbl_exprs = (List.map (fun expr -> s_check_expr f_context v_context expr) exprs) in
          if List.length (List.filter (fun a -> (
                        match a.exp with
                        Lval(lval) -> let lval_type = type_of_expr f_context v_context lval in (
                                if (List.length lval_type.s_dimension) > 0 then true
                                else false
                        )
                        | ArrayLit(_) -> false
                        | _ -> raise (Tbl_err (string_of_expr a.exp)))) tbl_exprs
                ) == (List.length exprs) then
      {exp = in_exp; typ = { s_ptype = Tbl; s_dimension = [] }}
    else
        raise (Tbl_err "Parameter in the table cannot be a type of array")

and s_check_rec f_context v_context in_exp exprs =
  let rec_ref_list = (List.map (fun expr -> s_check_expr f_context v_context expr) exprs) in
    if List.length (List.filter (fun a -> (match a.exp with RecRef(_,_) -> true | _ -> false))
rec_ref_list) == (List.length exprs) then
        { exp = Rec(List.map (fun a -> (s_check_expr f_context v_context a).exp) exprs); typ =
(type_of_expr f_context v_context in_exp)}
    else
        raise Rec_err

and s_check_rec_ref f_context v_context in_exp id expr =
        { exp = RecRef(id, (s_check_expr f_context v_context expr).exp); typ = type_of_expr f_context
v_context in_exp }

and s_check_fld f_context v_context in_exp expr id =
        let exp_type = type_of_expr f_context v_context expr in
                if List.length exp_type.s_dimension > 0 then
                        { exp = Fld((s_check_expr f_context v_context expr).exp, id); typ =
type_of_expr f_context v_context in_exp }

                else
                        raise Fld_err

and s_check_array f_context v_context in_exp id indices =
        let index_type = type_of_expr f_context v_context indices and
                        index_expr = s_check_expr f_context v_context indices in
                if List.length index_type.s_dimension == 0 then
                        {exp = Array(id, index_expr.exp); typ = type_of_expr f_context v_context
in_exp}
                else
```

```
                        match index_expr.exp with
                                Range(exp1, exp2) ->
                                        let type1 = type_of_expr f_context v_context exp1 and
                                                        type2 = type_of_expr f_context v_context exp2
in
                                                if type1.s_ptype == Int && type2.s_ptype == Int then
                                                        {exp = Array(id, index_expr.exp); typ =
type_of_expr f_context v_context in_exp}
                                                else if type1.s_ptype == Void && type2.s_ptype == Int
then
                                                        let start_range = {exp = Range(IntLit(0),
(s_check_expr f_context v_context exp2).exp); typ = type_of_expr f_context v_context in_exp } in
                                                                {exp = Array(id, start_range.exp); typ
= type_of_expr f_context v_context in_exp}
                                                else if type1.s_ptype == Int && type2.s_ptype == Void
then
                                                        let end_range = {exp = Range((s_check_expr
f_context v_context exp1).exp, IntLit(0)); typ = type_of_expr f_context v_context in_exp } in
                                                                {exp = Array(id, end_range.exp); typ =
type_of_expr f_context v_context in_exp}
                                                else raise Arr_err
                                | _ -> raise Arr_err

let s_check_var_type f_context v_context vtype =
    let dimention_type_list = (List.map (fun expr -> type_of_expr f_context v_context expr)
vtype.dimension) in
      if List.length (List.filter (fun a -> (a.s_ptype == Int)) dimention_type_list) == List.length
dimention_type_list
      then
          {s_ptype = vtype.ptype;
          s_dimension = List.map (fun expr -> s_check_expr f_context v_context expr) vtype.dimension }
      else (* Error dimension not int *)
          raise Var_type_err

let s_stmt_context_v f_context v_context level stmt = match stmt with
        VarDeclStmt(vdecl) ->
                let lhs = (s_check_var_type f_context v_context vdecl.vtype) and
                                rhs = (type_of_expr f_context v_context vdecl.vinit) in
                        (* check if variable is not initialized or is initialized to same type *)
                        if vdecl.vinit == Noexpr || vdecl.vinit == None || (List.length lhs.s_dimension
== List.length rhs.s_dimension && lhs.s_ptype == rhs.s_ptype) then
                                if StringMap.mem vdecl.vname v_context then
                                        let p_level = snd (StringMap.find vdecl.vname v_context) in
                                        (* check if variable has been defined in the same level *)
                                        if p_level == level then
                                                raise Duplicate_variable_err
                                        else
                                                StringMap.add vdecl.vname ((s_check_var_type f_context
v_context vdecl.vtype), level) v_context
                                else
                                        StringMap.add vdecl.vname ((s_check_var_type f_context
v_context vdecl.vtype), level) v_context
                        else
                                raise (Init_type_err ("lhs type: " ^ string_of_datatype lhs.s_ptype ^
", dimension: " ^ string_of_int (List.length lhs.s_dimension) ^ ", rhs type: " ^ string_of_datatype
rhs.s_ptype ^ ", dimension: " ^ string_of_int (List.length rhs.s_dimension)))
        | _ -> v_context
```

```
let s_check_var_decl f_context v_context vdecl =
        let lhs = (s_check_var_type f_context v_context vdecl.vtype) and
                        rhs = (type_of_expr f_context v_context vdecl.vinit) in
                if vdecl.vinit == Noexpr || vdecl.vinit == None || (List.length lhs.s_dimension ==
List.length rhs.s_dimension && lhs.s_ptype == rhs.s_ptype) then
                        {
                                s_vname = vdecl.vname;
                                s_vtype = (s_check_var_type f_context v_context vdecl.vtype);
                                s_vinit = (s_check_expr f_context v_context vdecl.vinit)
                        }
                else
                        raise (Init_type_err "Type does not match")

let rec s_check_stmt_list context_list stmt_list = match context_list, stmt_list with
     [], [] -> []
   | context_hd::context_tl, stmt_hd::stmt_tl ->
           (s_check_stmt (fst context_hd) (snd context_hd) stmt_hd)
        :: (s_check_stmt_list context_tl stmt_tl)
   | _, _ -> raise Stmt_list_err

and s_check_stmt f_context v_context level stmt = match stmt with
        Expr(expr) -> S_Expr(s_check_expr f_context v_context expr)
        | Return(expr) ->
                let t_expr = s_check_expr f_context v_context expr in
                        if StringMap.mem "0current" f_context then
                                let curr = StringMap.find "0current" f_context in
                                        if t_expr.typ.s_ptype == (snd (List.hd curr)).s_ptype &&
                                                                List.length t_expr.typ.s_dimension ==
List.length (snd (List.hd curr)).s_dimension then
                                                S_Return(s_check_expr f_context v_context expr)
                                        else
                                                raise Return_type_err
                        else
                                raise Current_not_found
        | Block(stmt_list) ->
                let first(f,_,_) = f and second(_,s,_) = s and third (_,_,t) = t in
                        S_Block(List.rev
                                (first
                                        (List.fold_left
                                                (fun x y -> (((s_check_stmt (second x) (third x)
(level+1) y) :: (first x)),

(second x),

(s_stmt_context_v (second x) (third x) (level+1) y)
                                                                                                )
                                                ) ([], f_context, v_context) stmt_list
                )
                ))
        | If(expr, stmt1, stmt2) ->
          let exp_type = type_of_expr f_context v_context expr in
                        if (exp_type.s_ptype == Bool && exp_type.s_dimension == []) then
                                S_If(s_check_expr f_context v_context expr,
                                s_check_stmt f_context v_context level stmt1,
                                s_check_stmt f_context v_context level stmt2)
                        else
```

```
                            raise Err_s_check_stmt_if;
        | For(expr1, expr2, expr3, stmt) ->
               let expr2_t = type_of_expr f_context v_context expr2 in
           if expr2_t.s_ptype ==  Bool && List.length expr2_t.s_dimension == 0
           then
               S_For(s_check_expr f_context v_context expr1,
               s_check_expr f_context v_context expr2,
               s_check_expr f_context v_context expr3,
               s_check_stmt f_context v_context level stmt)
           else
               raise Err_s_check_stmt_for;
        | While(expr, stmt) ->
              let expr_t = type_of_expr f_context v_context expr in
           if expr_t.s_ptype == Bool && expr_t.s_dimension == []
           then
               S_While(s_check_expr f_context v_context expr,
               s_check_stmt f_context v_context level stmt)
           else
               raise Err_s_check_stmt_while;
        | VarDeclStmt(vdecl) ->
              S_VarDeclStmt(s_check_var_decl f_context v_context vdecl)
        | Continue -> S_Continue
        | Break -> S_Break
        | EmptyStmt -> S_EmptyStmt

let s_check_func_decl f_context v_context fdecl =
        let s_formals = List.map (fun var_decl -> s_check_var_decl f_context v_context var_decl)
fdecl.formals in
        let s_return_type = s_check_var_type f_context v_context fdecl.return_type in
                {
                        s_fname = fdecl.fname;
                        s_formals = s_formals;
                        s_return_type = s_return_type;
                        s_body = s_check_stmt (StringMap.add "0current" [(List.map (fun decl ->
decl.s_vtype) s_formals, s_return_type)] f_context)

        (List.fold_left (fun map decl ->

                (* check if variable is already defined within the function *)

                if StringMap.mem decl.s_vname map && snd (StringMap.find decl.s_vname map) == 1 then

                        raise Duplicate_variable_err

                else

                        StringMap.add decl.s_vname (decl.s_vtype, 1) map) v_context s_formals)

        1 fdecl.body
                }

(* check list of function declarations *)
let rec s_check_func_decls f_context v_context func_decl_list = match func_decl_list with
        [] -> []
        | hd::tl ->
                (s_check_func_decl f_context v_context hd) :: (s_check_func_decls f_context v_context
tl)
```

```
let s_var_decl_to_var_map map s_vdecl =
        if StringMap.mem s_vdecl.s_vname map then
                (* check if variable name is already defined *)
                raise Duplicate_variable_err
        else
                (* add variable to map, 0 for gdecl *)
                StringMap.add s_vdecl.s_vname (s_vdecl.s_vtype, 0) map

(* check if vartiable type is the same for the two arguments *)
let is_s_var_type_equal t1 t2 =
        if t1.s_ptype == t2.s_ptype && List.length t1.s_dimension == List.length t2.s_dimension then
                true
        else
                false

(* recursively check if the variables in the list are equal *)
let rec is_s_var_type_list_equal l1 l2 = match l1, l2 with
        [], [] -> true
        | [], hd::tl -> false
        | hd::tl, [] -> false
        | h1::t1, h2::t2 ->
                if is_s_var_type_equal h1 h2 then
                        is_s_var_type_list_equal t1 t2
                else
                        false

(* check if the function return type is equal *)
let func_decl_check_func_map s_vtype_list_map s_vtype_list =
        let check = List.map (fun l1 -> is_s_var_type_list_equal (fst l1) s_vtype_list)
s_vtype_list_map in
                if List.length (List.filter (fun a -> a) check) != 0 then
                        true
                else
                        false

let func_decl_to_func_map map fdecl v_context =
        let func_s_vtype_list = List.map (fun decl -> decl.s_vtype) (List.map (fun arg ->
s_check_var_decl StringMap.empty v_context arg) fdecl.formals) in
                if StringMap.mem fdecl.fname map then
                        if func_decl_check_func_map (StringMap.find fdecl.fname map) func_s_vtype_list
then
                                (* raise duplicate function error if function of same name and return
type is already defined *)
                                raise (Duplicate_function_err fdecl.fname)
                        else
                                (* add function to map if function type is different *)
                                StringMap.add fdecl.fname ((func_s_vtype_list, s_check_var_type
StringMap.empty v_context fdecl.return_type)::StringMap.find fdecl.fname map) map
                else
                        (* if function name is not in the map, add new map with list of functions for
that fname *)
                        StringMap.add fdecl.fname [(func_s_vtype_list, s_check_var_type StringMap.empty
v_context fdecl.return_type)] map

let check prog check_option =
```

```
        let s_gdecls = List.map (fun var_decl -> s_check_var_decl StringMap.empty StringMap.empty
var_decl) prog.gdecls
        and extern_funs = (
                        let map = StringMap.empty in
                        let map = StringMap.add "print" [([{s_ptype = String; s_dimension = []}],
{s_ptype = Void; s_dimension = []});

                                                ([{s_ptype = Int; s_dimension = []}], {s_ptype = Void;
s_dimension = []});

                                                ([{s_ptype = Float; s_dimension = []}], {s_ptype =
Void; s_dimension = []});
                                                                                ([{s_ptype = Bool;
s_dimension = []}], {s_ptype = Void; s_dimension = []});
                                                                                ([{s_ptype = Rec;
s_dimension = []}], {s_ptype = Void; s_dimension = []});
                                                                                ([{s_ptype = Fld;
s_dimension = []}], {s_ptype = Void; s_dimension = []});
                                                                                ([{s_ptype = Tbl;
s_dimension = []}], {s_ptype = Void; s_dimension = []})] map in
                        let map = StringMap.add "get_int" [([{s_ptype = Fld; s_dimension = []};
{s_ptype = Int; s_dimension = []}], {s_ptype = Int; s_dimension = []})] map in
                        let map = StringMap.add "get_string" [([{s_ptype = Fld; s_dimension = []};
{s_ptype = Int; s_dimension = []}], {s_ptype = String; s_dimension = []})] map in
                        let map = StringMap.add "get_float" [([{s_ptype = Fld; s_dimension = []};
{s_ptype = Int; s_dimension = []}], {s_ptype = Float; s_dimension = []})] map in
                        let map = StringMap.add "get_bool" [([{s_ptype = Fld; s_dimension = []};
{s_ptype = Int; s_dimension = []}], {s_ptype = Bool; s_dimension = []})] map in
                let map = StringMap.add "load" [([{s_ptype = String; s_dimension = []}], {s_ptype =
Tbl; s_dimension = []})] map in
                let map = StringMap.add "save" [([{s_ptype = Tbl; s_dimension = []}; {s_ptype =
String; s_dimension = []}], {s_ptype = Void; s_dimension = []})] map in
                let map = StringMap.add "access" [([{s_ptype = Tbl; s_dimension = []}; {s_ptype =
String; s_dimension = [{ exp = StringLit(""); typ = {s_ptype = String; s_dimension = []}}]}; {s_ptype =
Int; s_dimension = []}], {s_ptype = Tbl; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = Int;
s_dimension = []}; {s_ptype = Int; s_dimension = []}], {s_ptype = Tbl; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = String;
s_dimension = [{ exp = StringLit(""); typ = {s_ptype = String; s_dimension = []}}]}], {s_ptype = Tbl;
s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = String;
s_dimension = []}], {s_ptype = Fld; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = Int;
s_dimension = []}], {s_ptype = Rec; s_dimension = []})] map in
                        let map = StringMap.add "append" [([{s_ptype = Tbl; s_dimension = []}; {s_ptype
= Rec; s_dimension = []}], {s_ptype = Tbl; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = Fld;
s_dimension = []}], {s_ptype = Tbl; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = Tbl;
s_dimension = []}; {s_ptype = Bool; s_dimension = []}], {s_ptype = Tbl; s_dimension = []})] map in
```

```
                let map = StringMap.add "modify" [([{s_ptype = Tbl; s_dimension = []}; {s_ptype =
Int; s_dimension = []}; {s_ptype = Int; s_dimension = []}; {s_ptype = Float; s_dimension = []}],
{s_ptype = Tbl; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = Int;
s_dimension = []}; {s_ptype = Int; s_dimension = []}; {s_ptype = Bool; s_dimension = []}], {s_ptype =
Tbl; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = Int;
s_dimension = []}; {s_ptype = Int; s_dimension = []}; {s_ptype = String; s_dimension = []}], {s_ptype =
Tbl; s_dimension = []});

                                                ([{s_ptype = Tbl; s_dimension = []}; {s_ptype = Int;
s_dimension = []}; {s_ptype = Int; s_dimension = []}; {s_ptype = Int; s_dimension = []}], {s_ptype =
Tbl; s_dimension = []})] map in
                let map = StringMap.add "convert" [([{s_ptype = Tbl; s_dimension = []}; {s_ptype =
Int; s_dimension = []}; {s_ptype = String; s_dimension = []}], {s_ptype = Tbl; s_dimension = []})] map
in
                let map = StringMap.add "set_int" [([{s_ptype = Fld; s_dimension = []}; {s_ptype =
Int; s_dimension = []}; {s_ptype = Int; s_dimension = []}], {s_ptype = Int; s_dimension = []})] map in
                        let map = StringMap.add "set_string" [([{s_ptype = Fld; s_dimension = []};
{s_ptype = Int; s_dimension = []}; {s_ptype = String; s_dimension = []}], {s_ptype = String;
s_dimension = []})] map in
                        let map = StringMap.add "set_float" [([{s_ptype = Fld; s_dimension = []};
{s_ptype = Int; s_dimension = []}; {s_ptype = Float; s_dimension = []}], {s_ptype = Float; s_dimension
= []})] map in
                        let map = StringMap.add "set_bool" [([{s_ptype = Fld; s_dimension = []};
{s_ptype = Int; s_dimension = []}; {s_ptype = Bool; s_dimension = []}], {s_ptype = Bool; s_dimension =
[]})] map in
                (* add dave library functions *)
                if check_option = "import" then
                        map
                else
                        let map = StringMap.add "min_value" [([{s_ptype = Fld; s_dimension = []}],
{s_ptype = Float; s_dimension = []})] map in
                                let map = StringMap.add "max_value" [([{s_ptype = Fld; s_dimension =
[]}], {s_ptype = Float; s_dimension = []})] map in
                                  let map = StringMap.add "mean_value" [([{s_ptype = Fld; s_dimension = []}],
{s_ptype = Float; s_dimension = []})] map in
                                StringMap.add "median_value"            [([{s_ptype = Fld; s_dimension
= []}], {s_ptype = Float; s_dimension = []})] map
                )
                in {
                        s_gdecls = s_gdecls;
                        s_fdecls =
                                let v_context = List.fold_left s_var_decl_to_var_map StringMap.empty
s_gdecls in
                                let f_context = List.fold_left (fun ext_func_map func_decl ->
func_decl_to_func_map ext_func_map func_decl v_context) extern_funs prog.fdecls in
                                        if (StringMap.mem "main" f_context) || check_option = "import"
then
                                                s_check_func_decls f_context v_context prog.fdecls
                                        else
                                        (* main has to be defined in the function *)
                                                raise Main_not_found_err
                }
```

semanticExceptions.ml

```
exception Main_not_found_err
exception Duplicate_function_err of string
exception Duplicate_variable_err
exception Stmt_list_err
exception Not_implemented_err
exception Type_err of string
exception Arr_err
exception Access_err
exception Var_type_err
exception Init_type_err of string
exception No_func_err of string
exception No_variable_err of string
exception Return_type_err
exception Current_not_found
exception Err_s_check_stmt_if
exception Err_s_check_stmt_for
exception Err_s_check_stmt_while
exception Invalid_type
exception Invalid_statement
exception Invalid_func_err of string
exception Array_lit_err
exception Fld_err
exception Rec_err
exception Tbl_err of string
exception Rec_ref_err
exception Compile_err of string
exception Cast_err of string
```

compile.ml

```
open Ast
open Sast
open SemanticExceptions
open Printf

let rec string_of_datatype = function
  Int -> "int"
| Float -> "double"
| Bool -> "bool"
| String -> "string"
| Tbl -> "tbl"
| Rec -> "rec"
| Fld -> "fld"
| Void -> "void"

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

let string_of_postop = function
  Inc -> "++"
```

```
| Dec -> "--"

let string_of_binop = function
  Add -> "+"
| Sub -> "-"
| Mul -> "*"
| Div -> "/"
| Mod -> "%"
| Exp -> "^"
| Eq -> "=="
| Neq -> "!="
| Lt -> "<"
| Leq -> "<="
| Gt -> ">"
| Geq -> ">="
| Or -> "||"
| And -> "&&"

let string_of_asnop = function
  Asn -> "="
| Addeq -> "+="
| Subeq -> "-="
| Muleq -> "*="
| Diveq -> "/="
| Modeq -> "%="

let rec list_of_dim v = match v with
  [] -> ""
  | hd::tl ->
    let exp_string = string_of_expr hd.exp in
      if exp_string = "0" then "" else "(" ^ exp_string ^ ")"

and string_of_expr = function
  Range(exp1, exp2) -> "range(" ^ string_of_expr exp1 ^ ", " ^ string_of_expr exp2 ^ ")"
| IntLit(lit) -> string_of_int lit
| FloatLit(lit) -> string_of_float lit
| StringLit(lit) -> "\"" ^ lit ^ "\""
| BoolLit(lit) -> string_of_bool lit
| ArrayLit(exps) -> "{" ^ String.concat ", " (List.map string_of_expr exps) ^ "}"
| Binop(exp1, binop, exp2) -> (
  match exp1, exp2 with
    StringLit(lit1), StringLit(lit2) -> "append(\"" ^ lit1 ^ "\",\"" ^ lit2 ^ "\")"
    | _,_ -> "(" ^ string_of_expr exp1 ^ " " ^ string_of_binop binop ^ " " ^ string_of_expr exp2 ^ ")"
  )
| Unop(unop, exp) -> "(" ^ string_of_unop unop ^ string_of_expr exp ^ ")"
| Postop(lvalue, postop) -> "(" ^ string_of_expr lvalue ^ string_of_postop postop ^ ")"
| AssignOp(lvalue, asnop, exp) -> "(" ^ string_of_expr lvalue ^ " " ^ string_of_asnop asnop ^ " " ^
string_of_expr exp ^ ")"
| Cast(datatype, exp) -> "((" ^ string_of_datatype datatype.ptype ^ ")" ^ string_of_expr exp ^ ")"
| FuncCall(id, exps) -> string_of_func_call id exps
| Tbl(exps) -> "tbl(" ^ String.concat ", " (List.map string_of_expr exps) ^ ")"
| Rec(exps) -> "rec(" ^ String.concat ", " (List.map string_of_expr exps) ^ ")"
| RecRef(id, exp) -> "tuple(" ^ string_of_expr exp ^ ", \"" ^ id ^ "\")"
| Fld(exp, lit) -> "fld(" ^ string_of_expr exp ^ ", \"" ^ lit ^ "\")"
| Lval(lvalue) -> string_of_expr lvalue
| Noexpr -> ""
| None -> "NULL"
```

```
| Var(exp) -> exp
| Array(exp1, exp2) -> string_of_array exp1 exp2
| Access(exp1, exp2) -> "(" ^ string_of_expr exp1 ^ "." ^ exp2 ^ ")"

and string_of_array id exp2 = match exp2 with
  | Range(id1, id2) ->
    if string_of_expr id2 = "0" || string_of_expr id2 = "" then
      "slice_array(" ^ id ^ ", " ^ string_of_expr id1 ^ ", (int)" ^ id ^ ".size())"
    else if string_of_expr id1 = "" then
      "slice_array(" ^ id ^ ", 0, " ^ string_of_expr id2 ^ ")"
    else
      "slice_array(" ^ id ^ ", " ^ string_of_expr id1 ^ ", " ^ string_of_expr id2 ^ ")"
  | _ -> "(" ^ id ^ "[" ^ string_of_expr exp2 ^ "])"

and string_of_func_call id exps = id ^ "(" ^ String.concat ", " (List.map string_of_expr exps) ^ ")"

and string_of_vtype v =
  let dimension = v.s_dimension in
    match dimension with
    [] -> string_of_datatype v.s_ptype
    | _ -> "vector<" ^ string_of_datatype v.s_ptype ^ "> " ^ list_of_dim (List.rev v.s_dimension)

and string_of_var v id =
  let dimension = v.s_dimension in
    match dimension with
    [] -> string_of_datatype v.s_ptype ^ " " ^ id
    | _ -> "vector<" ^ string_of_datatype v.s_ptype ^ "> " ^ id ^ list_of_dim (List.rev v.s_dimension)

let string_of_decl vdecl =
  let init = vdecl.s_vinit.exp in
  match init with
    Noexpr -> ((string_of_var vdecl.s_vtype vdecl.s_vname))
    | Lval(exp) -> (match exp with
        Array(id, exp1) -> (match exp1 with
            Range(id1, id2) -> string_of_var vdecl.s_vtype vdecl.s_vname ^ " = _slice_array(" ^ id ^ ",
" ^ string_of_expr id1 ^ ", " ^ string_of_expr id2 ^ ")"
          | _ -> (string_of_var vdecl.s_vtype vdecl.s_vname) ^ " = " ^ string_of_expr
vdecl.s_vinit.exp
         )
       | _ -> (string_of_var vdecl.s_vtype vdecl.s_vname) ^ " = " ^ string_of_expr vdecl.s_vinit.exp
      )
    | Rec(exprs) -> "tuple _" ^ vdecl.s_vname ^ "[] = {" ^ (String.concat ", " (List.map string_of_expr
exprs)) ^ "};\n" ^
                    (string_of_var vdecl.s_vtype vdecl.s_vname) ^ " = rec(_" ^ vdecl.s_vname ^ ",
getArrayLen(_" ^ vdecl.s_vname ^ "))"
    | Fld(expr, id) ->
      (match expr with
        ArrayLit(array_expr) -> (match array_expr with
          _ ->  let array_type = (match (List.hd array_expr) with StringLit(_) -> "string" | BoolLit(_)
-> "bool" | FloatLit(_) -> "double" | _ -> "int") in
                array_type ^ " __" ^ vdecl.s_vname ^ "[] = " ^ string_of_expr expr ^ ";\n" ^
                "vector<" ^ array_type ^ "> _" ^ vdecl.s_vname ^ " = to_vector(__" ^ vdecl.s_vname ^ ",
getArrayLen(__" ^ vdecl.s_vname ^ "));\n" ^
                "fld " ^ vdecl.s_vname ^ " = fld(&_" ^ vdecl.s_vname ^ "[0],\"" ^ id ^ "\", _" ^
vdecl.s_vname ^".size())"
        )
```

```
       | _ -> "fld " ^ vdecl.s_vname ^ " = fld(&" ^ string_of_expr expr ^ "[0], \"" ^ id ^ "\", " ^
string_of_expr expr ^".size())"
      )
    | Tbl(exprs) ->
        let tbl_element = string_of_expr (List.hd exprs) in
            "tbl " ^ vdecl.s_vname ^ " = tbl(&" ^ tbl_element ^ "[0], " ^ tbl_element ^ ".size(), " ^
tbl_element ^ "[0].length)"
    | ArrayLit(exprs) ->
      let array_type = (match vdecl.s_vinit.typ.s_ptype with Rec -> "rec" | Fld -> "fld" | String ->
"string" | Bool -> "bool" | Float -> "double" | _ -> "int") in
        array_type ^ " _" ^ vdecl.s_vname ^ "[] = {" ^ (String.concat ", " (List.map string_of_expr
exprs)) ^ "};\n" ^
        "vector<" ^ array_type ^ "> " ^ vdecl.s_vname ^ " = to_vector(_" ^ vdecl.s_vname ^ ",
getArrayLen(_" ^ vdecl.s_vname ^ "))"
    | _ -> (string_of_var vdecl.s_vtype vdecl.s_vname) ^ " = " ^ string_of_expr vdecl.s_vinit.exp

let rec gen_stmt = function
  S_Expr(exp) -> "(" ^ (string_of_expr exp.exp) ^ ");"
| S_Return(exp) -> "return " ^ string_of_expr exp.exp ^ ";"
| S_Block(stmt_list) -> "{\n" ^ (String.concat "\n" (List.map gen_stmt stmt_list)) ^ "\n}\n"
| S_If(exp, stmt1, stmt2) -> (if stmt2 == S_EmptyStmt then
  "if (" ^ (string_of_expr exp.exp) ^ ")\n" ^ (gen_stmt stmt1) else
  "if (" ^ (string_of_expr exp.exp) ^ ")\n" ^ (gen_stmt stmt1) ^ "else " ^ (gen_stmt stmt2))
| S_For(init, test, after, stmt) -> "for (" ^ string_of_expr init.exp ^ "; " ^ string_of_expr test.exp
^ "; " ^ string_of_expr after.exp ^ ") " ^ gen_stmt stmt
| S_While(test, stmt) -> "while (" ^ (string_of_expr test.exp) ^ ") " ^ (gen_stmt stmt)
| S_VarDeclStmt(decl) -> string_of_decl decl ^ ";"
| S_Continue -> "continue;"
| S_Break -> "break;"
| S_EmptyStmt -> ";"

let string_of_func_decl funcdecl =
  string_of_vtype funcdecl.s_return_type ^ " "
  ^ funcdecl.s_fname ^ "("
  ^ (String.concat ", " (List.map string_of_decl funcdecl.s_formals))
  ^ ") {\n"
  ^ gen_stmt funcdecl.s_body ^ "\n}"

let string_of_program prg =
  (String.concat "\n" (List.map string_of_decl prg.s_gdecls)) ^ "\n"
  ^ (String.concat "\n" (List.map string_of_func_decl prg.s_fdecls)) ^ "\n"

let compile oc prg =
  let out_file = open_out oc in
  fprintf out_file "#include \"dave.h\"\n";
  fprintf out_file "%s" (string_of_program (List.hd prg));
  fprintf out_file "%s" (string_of_program (List.nth prg 1));
  close_out out_file
```

dave.h

```
#define __DAVE_H_
#include "dave_io.hpp"
```

```
void print(int n) {
        cout << n << endl;
}

void print(double d) {
        cout << d << endl;
}

void print(string str) {
        cout << str << endl;
}

void print(const char *str) {
        cout << str << endl;
}

void print(bool b) {
        cout << (b ? "true" : "false") << endl;
}

void print(rec r) {
        cout << r << endl;
}

void print(fld f) {
        cout << f << endl;
}

void print(tbl t) {
        cout << t << endl;
}

string append(string a, string b) {
        return a + b;
}

void set_int(fld a, int index, int b) {
        a.f_int[index] = b;
}

void set_float(fld a, int index, float b) {
        a.f_double[index] = b;
}

void set_bool(fld a, int index, bool b) {
        a.f_bool[index] = b;
}

void set_string(fld a, int index, string b) {
        a.f_string[index] = b;
}

int get_int(fld a, int index) {
        return a.f_int[index];
}

double get_float(fld a, int index) {
```

```
        return a.f_double[index];
}

string get_string(fld a, int index) {
        return a.f_string[index];
}

bool get_bool(fld a, int index) {
        return a.f_bool[index];
}

vector<int> to_vector(int arr[], int length) {
        vector<int> v;
        for (int i = 0; i < length; i++) {
                v.push_back(arr[i]);
        }
        return v;
}

vector<double> to_vector(double arr[], int length) {
        vector<double> v;
        for (int i = 0; i < length; i++) {
                v.push_back(arr[i]);
        }
        return v;
}

vector<bool> to_vector(bool arr[], int length) {
        vector<bool> v;
        for (int i = 0; i < length; i++) {
                v.push_back(arr[i]);
        }
        return v;
}

vector<string> to_vector(string arr[], int length) {
        vector<string> v;
        for (int i = 0; i < length; i++) {
                v.push_back(arr[i]);
        }
        return v;
}

vector<fld> to_vector(fld arr[], int length) {
        vector<fld> v;
        for (int i = 0; i < length; i++) {
                v.push_back(arr[i]);
        }
        return v;
}

vector<rec> to_vector(rec arr[], int length) {
        vector<rec> v;
        for (int i = 0; i < length; i++) {
                v.push_back(arr[i]);
        }
        return v;
```

```
}

vector<int> slice_array(vector<int> src, int begin, int end) {
	int j = 0;
	for (int i = begin; i < end; i++) {
		src[j++] = src[i];
	}
	src.resize(end-begin);
	return src;
}

vector<double> slice_array(vector<double> src, int begin, int end) {
	int j = 0;
	for (int i = begin; i < end; i++) {
		src[j++] = src[i];
	}
	src.resize(end-begin);
	return src;
}

vector<string> slice_array(vector<string> src, int begin, int end) {
	int j = 0;
	for (int i = begin; i < end; i++) {
		src[j++] = src[i];
	}
	src.resize(end-begin);
	return src;
}

vector<bool> slice_array(vector<bool> src, int begin, int end) {
	int j = 0;
	for (int i = begin; i < end; i++) {
		src[j++] = src[i];
	}
	src.resize(end-begin);
	return src;
}

vector<fld> slice_array(vector<fld> src, int begin, int end) {
	vector<fld> dest;
	int j = 0;
	for (int i = begin; i < end; i++) {
		dest.push_back(src[i]);
	}
	return dest;
}

vector<rec> slice_array(vector<rec> src, int begin, int end) {
	vector<rec> dest;
	int j = 0;
	for (int i = begin; i < end; i++) {
		dest.push_back(src[i]);
	}
	return dest;
}
```

dave_io.hpp

```
#include "dave.hpp"
#include <fstream>
using namespace std;

tbl load(string filename) {
    char buffer[256];
    ifstream in(filename);
    bool flag = false;
    vector< rec > storage;
    vector< string > name;
    vector< string > element;
    vector< tuple > tuples;

    if (! in.is_open()) {
        cout << "File opening errorness" << endl;
        exit(1);
    }
    while (! in.eof()) {
        if (!flag) {
            in.getline(buffer, 256, '\n');
            string whole = string(buffer);
            int i = 0;
            int j = 0;
            while (buffer[i+j] != ';') {
                // note here, every line need to be end with " ;"
                if (buffer[i+j] == '\t') {
                    string temp = whole.substr(i, j);
                    name.push_back(temp);
                    i = i+j+1;
                    j = 0;
                    continue;
                }
                j++;
            }
            flag = true;
            continue;
        }
        element.clear();
        tuples.clear();
        in.getline(buffer, 256, '\n');
        string whole = string(buffer);
        int i = 0;
        int j = 0;
        while (buffer[i+j] != ';') {
            if (buffer[i+j] == '\t') {
                string temp = whole.substr(i, j);
                element.push_back(temp);
                i = i+j+1;
                j = 0;
                continue;
            }
            j++;
```

```
        }
        for (int k=0; k<name.size(); k++) {
            tuple newtup = tuple(element[k], name[k]);
            tuples.push_back(newtup);
        }
        rec newrec = rec(&tuples[0], tuples.size());
        storage.push_back(newrec);
    }
    in.close();
    tbl readtbl = tbl(&storage[0], storage.size(), storage[0].length);
    return readtbl;
}

void save(tbl &in, string filename) {
    ofstream out;
    out.open(filename);
    for (int i=0; i<in.row_length; i++) {
        out << in.t[i].name << "\t";
    }
    out << ";" << endl;
    for (int i=0; i<in.col_length; i++) {
        for (int j=0; j<in.row_length; j++) {
            if ( in.t[j].type == 0 ) {
                out << in.t[j].f_int[i];
            } else if ( in.t[j].type == 1 ) {
                out << in.t[j].f_double[i];
            } else if ( in.t[j].type == 2 ) {
                out << in.t[j].f_string[i];
            } else if ( in.t[j].type == 3 ) {
                if (in.t[j].f_bool[i] == 0) {
                    out << "false";
                } else {
                    out << "true";
                }
            }
            out << "\t";
        }
        if (i != in.col_length-1) {
            out << ";" << endl;
        } else {
            out << ";";
        }
    }
    out.close();
}

/*int main(int argc, char const *argv[]) {
        tbl test = tbl_read(argv[1]);
        cout << test;
    tbl_write(test, argv[2]);
    test = tbl_read(argv[2]);
    cout << test;
        return 0;
}*/
```

dave.hpp

```
/* #ifndef _DAVE_HPP_*/
#define _DAVE_HPP_
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <stdexcept>

using namespace std;

template <class T>
int getArrayLen(T& array)
{
    return (sizeof(array)/sizeof(array[0]));
}

class tuple {
public:
    string name;
    int type;
    int content_int;
    double content_double;
    bool content_bool;
    string content_string;
        tuple(int initial,string str) {
        type = 0;
        content_int = initial;
        name = str;
    }
    tuple(double initial,string str) {
        type = 1;
        content_double = initial;
        name = str;
    }
    tuple(string initial,string str) {
        if (str.compare("true") == 0) {
            type = 3;
            content_bool = true;
            name = str;
        } else if (str.compare("false") == 0) {
            type = 3;
            content_bool = false;
            name = str;
        } else {
            type = 2;
            content_string = initial;
            name = str;
        }
    }
    /*tuple(bool initial,string str) {
        type = 3;
        content_bool = initial;
        name = str;
    }*/
```

```
};

class rec {
public:
        int length;
        vector< tuple > r;
    rec(tuple *array, int len) {
        length = len;
        for (int i=0; i<length; i++)
            r.push_back(array[i]);
    }
};

class fld {
public:
    int length;
    string name;
    int type;
    vector<int> f_int;
    vector<double> f_double;
    vector<string> f_string;
    vector<bool> f_bool;
    fld(int *array, string str, int len) {
        type = 0;
        length = len;
        name = str;
        for (int i=0; i<length; i++) {
            int var = array[i];
            f_int.push_back(var);
        }
    }
    fld(double *array, string str, int len) {
        type = 1;
        length = len;
        name = str;
        for (int i=0; i<length; i++)
            f_double.push_back(array[i]);
    }
    fld(string *array, string str, int len) {
        type = 2;
        length = len;
        name = str;
        for (int i=0; i<length; i++) {
            string var = array[i];
            f_string.push_back(var);
        }
    }
    fld(bool *array, string str, int len) {
        type = 3;
        length = len;
        name = str;
        for (int i=0; i<length; i++)
            f_bool.push_back(array[i]);
    }
};
```

```
class tbl {
public:
        int row_length;
    int col_length;
    vector<fld> t;
    tbl(fld *array, int row_len, int col_len) {
        col_length = col_len;
        row_length = row_len;
        for (int i=0; i<row_len; i++) {
            t.push_back(array[i]);
        }
    }
    tbl(rec *array, int col_len, int row_len) {
        col_length = col_len;
        row_length = row_len;
        for (int i=0; i<row_len; i++) {
            string name = array[0].r[i].name;
            if (array[0].r[i].type == 0) {
                vector<int> data;
                for(int j=0; j<col_len; j++) {
                    for (int k=0; k<row_len; k++) {
                        if (array[j].r[k].name.compare(name) == 0) {
                            data.push_back(array[j].r[k].content_int);
                        }
                    }
                }
                fld temp = fld (&data[0], name, col_len);
                t.push_back(temp);
            } else if (array[0].r[i].type == 1) {
                vector<double> data;
                for(int j=0; j<col_len; j++) {
                    for (int k=0; k<row_len; k++) {
                        if (array[j].r[k].name.compare(name) == 0) {
                            data.push_back(array[j].r[k].content_double);
                        }
                    }
                }
                fld temp = fld (&data[0], name, col_len);
                t.push_back(temp);
            } else if (array[0].r[i].type == 2) {
                vector<string> data;
                for(int j=0; j<col_len; j++) {
                    for (int k=0; k<row_len; k++) {
                        if (array[j].r[k].name.compare(name) == 0) {
                            data.push_back(array[j].r[k].content_string);
                        }
                    }
                }
                fld temp = fld (&data[0], name, col_len);
                t.push_back(temp);
            } else if (array[0].r[i].type == 3) {
                bool data[col_len];
                for(int j=0; j<col_len; j++) {
                    for (int k=0; k<row_len; k++) {
                        if (array[j].r[k].name.compare(name) == 0) {
                            data[j] = array[j].r[k].content_bool;
                        }
```

```
                }
            }
            fld temp = fld (data, name, col_len);
            t.push_back(temp);
        }
    }
}

tbl(tbl original, int col, int row, int newele) {
    if (col > original.col_length || row > original.row_length) {
        cout << "reset failed!" << endl;
        exit(1);
    }
    col_length = original.col_length;
    row_length = original.row_length;
    t = original.t;
    t[row].f_int[col] = newele;
}
tbl(tbl original, int col, int row, double newele) {
    if (col > original.col_length || row > original.row_length) {
        cout << "reset failed!" << endl;
        exit(1);
    }
    col_length = original.col_length;
    row_length = original.row_length;
    t = original.t;
    t[row].f_double[col] = newele;
}
tbl(tbl original, int col, int row, string newele) {
    if (col > original.col_length || row > original.row_length) {
        cout << "reset failed!" << endl;
        exit(1);
    }
    col_length = original.col_length;
    row_length = original.row_length;
    t = original.t;
    t[row-1].f_string[col-1] = newele;
}
/* tbl(tbl original, int col, int row, bool newele) {
    if (col > original.col_length || row > original.row_length) {
        cout << "reset failed!" << endl;
        exit(1);
    }
    col_length = original.col_length;
    row_length = original.row_length;
    t = original.t;
    t[row-1].f_bool[col-1] = newele;
}*/
};

tbl append(tbl original, fld newfld) {
    for (int j=0; j<original.row_length; j++) {
        if (newfld.name.compare(original.t[j].name) == 0) {
            cout << "the name has already existed" << endl;
            return original;
        }
    }
```

```
    original.row_length ++;
    original.t.push_back(newfld);
    return original;
}

tbl append(tbl original, rec newrec) {
    original.col_length ++;
    for (int i=0; i<newrec.length; i++) {
        for (int j=0; j<original.row_length; j++) {
            if (newrec.r[i].name.compare(original.t[j].name) == 0) {
                if (newrec.r[i].type == 0) {
                    original.t[j].f_int.push_back(newrec.r[i].content_int);
                } else if (newrec.r[i].type == 1) {
                    original.t[j].f_double.push_back(newrec.r[i].content_double);
                } else if (newrec.r[i].type == 2) {
                    original.t[j].f_string.push_back(newrec.r[i].content_string);
                } else if (newrec.r[i].type == 3) {
                    original.t[j].f_bool.push_back(newrec.r[i].content_bool);
                }
                break;
            }
        }
    }
    return original;
}

tbl append(tbl source1, tbl source2, bool ishorizonal) {
    if (ishorizonal) {
        if (source1.col_length != source2.col_length) {
            cout << "combine failed!" << endl;
            exit(1);
        }
        source1.row_length = source1.row_length + source2.row_length;
        for (int i=0; i<source2.row_length; i++) {
            source1.t.push_back(source2.t[i]);
        }
    } else {
        if (source1.row_length != source2.row_length) {
            cout << "combine failed!" << endl;
            exit(1);
        }
        source1.col_length = source1.col_length + source2.col_length;
        for (int i=0; i<source2.row_length; i++) {
            for (int j=0; j<source1.row_length; j++) {
                if (source2.t[i].name.compare(source1.t[j].name) == 0) {
                    for (int k=0; k< source2.t[i].length; k++) {
                        if (source2.t[i].type == 0) {
                            source1.t[j].f_int.push_back(source2.t[i].f_int[k]);
                        } else if (source2.t[i].type == 1) {
                            source1.t[j].f_double.push_back(source2.t[i].f_double[k]);
                        } else if (source2.t[i].type == 2) {
                            source1.t[j].f_string.push_back(source2.t[i].f_string[k]);
                        } else if (source2.t[i].type == 3) {
                            source1.t[j].f_bool.push_back(source2.t[i].f_bool[k]);
                        }
                    }
                    break;
```

```
                }
            }
        }
    }
    return source1;
}

tbl modify(tbl original, int col, int row, int newele) {
    if (original.t[row].type != 0) {
        cout << "type is wrong" << endl;
        exit(1);
    }
    if (col > original.col_length || row > original.row_length) {
        cout << "reset failed!" << endl;
        exit(1);
    }
    original.t[row].f_int[col] = newele;
    return original;
}
tbl modify(tbl original, int col, int row, double newele) {
    if (original.t[row].type != 1) {
        cout << "type is wrong" << endl;
        exit(1);
    }
    if (col > original.col_length || row > original.row_length) {
        cout << "reset failed!" << endl;
        exit(1);
    }
    original.t[row].f_double[col] = newele;
    return original;
}
tbl modify(tbl original, int col, int row, string newele) {
    if (col > original.col_length || row > original.row_length) {
        cout << "reset failed!" << endl;
        exit(1);
    }
    if (original.t[row].type == 2) {
        if (newele.compare("false") != 0 && newele.compare("true") != 0) {
            original.t[row].f_string[col] = newele;
        }
    } else if (original.t[row].type == 3) {
        if (newele.compare("false") == 0) {
            original.t[row].f_bool[col] = 0;
        } else if (newele.compare("true") == 0) {
            original.t[row].f_bool[col] = 1;
        }
    } else {
        cout << "type is wrong" << endl;
        exit(1);
    }
    return original;
}

tbl convert(tbl original, int row, string type) {
    int newtype;
    if (type.compare("int") == 0) {
        newtype = 0;
```

```
    } else if (type.compare("doulbe") == 0) {
        newtype = 1;
    } else if (type.compare("string") == 0) {
        newtype = 2;
    }
    if (original.t[row].type != newtype) {
        if (newtype == 2) {
            if (original.t[row].type == 0) {
                original.t[row].type = 2;
                for (int i=0; i<original.col_length; i++) {
                    stringstream strStream;
                    strStream << original.t[row].f_int[i];
                    string s = strStream.str();
                    original.t[row].f_string.push_back(s);
                }
            }
            if (original.t[row].type == 1) {
                original.t[row].type = 2;
                for (int i=0; i<original.col_length; i++) {
                    stringstream strStream;
                    strStream << original.t[row].f_double[i];
                    string s = strStream.str();
                    original.t[row].f_string.push_back(s);
                }
            }
        }
        else if (original.t[row].type == 2) {
            if (newtype == 0) {
                original.t[row].type = 0;
                for (int i=0; i<original.col_length; i++) {
                    char temp[10];
                    strcpy(temp, original.t[row].f_string[i].c_str());
                    original.t[row].f_int[i] = atoi(temp);
                }
            }
            if (newtype == 1) {
                original.t[row].type = 1;
                for (int i=0; i<original.row_length; i++) {
                    char temp[10];
                    strcpy(temp, original.t[row].f_string[i].c_str());
                    original.t[row].f_int[i] = atof(temp);
                }
            }
        } else if (newtype == 1) {
            original.t[row].type = 1;
            for (int i=0; i<original.row_length; i++) {
                original.t[row].f_double[i] = double(original.t[row].f_int[i]);
            }
        } else if (newtype == 0) {
            original.t[row].type = 0;
            for (int i=0; i<original.row_length; i++) {
                original.t[row].f_int[i] = int(original.t[row].f_double[i]);
            }
        }
    }
    return original;
}
```

```
tbl tbl_plus(tbl source1, tbl source2) {
    if (source1.col_length != source2.col_length ||
        source1.row_length != source2.row_length) {
        return source1;
    }
    for (int i=0; i<source1.col_length; i++) {
        for (int j=0; j<source1.row_length; j++) {
            if (source1.t[j].type == 0 && source2.t[j].type == 0) {
                source1.t[j].f_int[i] += source2.t[j].f_int[i];
            }
            else if (source1.t[j].type == 1 && source2.t[j].type == 1) {
                source1.t[j].f_double[i] += source2.t[j].f_double[i];
            }
        }
    }
    return source1;
}

tbl tbl_minus(tbl source1, tbl source2) {
    if (source1.col_length != source2.col_length ||
        source1.row_length != source2.row_length) {
        return source1;
    }
    for (int i=0; i<source1.col_length; i++) {
        for (int j=0; j<source1.row_length; j++) {
            if (source1.t[j].type == 0 && source2.t[j].type == 0) {
                source1.t[j].f_int[i] -= source2.t[j].f_int[i];
            }
            else if (source1.t[j].type == 1 && source2.t[j].type == 1) {
                source1.t[j].f_double[i] -= source2.t[j].f_double[i];
            }
        }
    }
    return source1;
}

tbl tbl_mult(tbl source1, tbl source2) {
    if (source1.col_length != source2.col_length ||
        source1.row_length != source2.row_length) {
        return source1;
    }
    for (int i=0; i<source1.col_length; i++) {
        for (int j=0; j<source1.row_length; j++) {
            if (source1.t[j].type == 0 && source2.t[j].type == 0) {
                source1.t[j].f_int[i] *= source2.t[j].f_int[i];
            }
            else if (source1.t[j].type == 1 && source2.t[j].type == 1) {
                source1.t[j].f_double[i] *= source2.t[j].f_double[i];
            }
        }
    }
    return source1;
}

tbl tbl_div(tbl source1, tbl source2) {
```

```
    if (source1.col_length != source2.col_length ||
        source1.row_length != source2.row_length) {
        return source1;
    }
    for (int i=0; i<source1.col_length; i++) {
        for (int j=0; j<source1.row_length; j++) {
            if (source1.t[j].type == 0 && source2.t[j].type == 0) {
                source1.t[j].f_int[i] /= source2.t[j].f_int[i];
            }
            else if (source1.t[j].type == 1 && source2.t[j].type == 1) {
                source1.t[j].f_double[i] /= source2.t[j].f_double[i];
            }
        }
    }
    return source1;
}

/*int access_int(rec source, string name) {
    int result;
    for (int i=0; i<source.length; i++) {
        if (source.r[i].name.compare(name) == 0) {
            result = source.r[i].content_int;
            return result;
        }
    }
    cout << "Not find the access."<<endl;
    return 0;
}

double access_double(rec source, string name) {
    double result;
    for (int i=0; i<source.length; i++) {
        if (source.r[i].name.compare(name) == 0) {
            result = source.r[i].content_double;
            return result;
        }
    }
    cout << "Not find the access."<<endl;
    return 0.0;
}

string access_string(rec source, string name) {
    string result;
    for (int i=0; i<source.length; i++) {
        if (source.r[i].name.compare(name) == 0) {
            result = source.r[i].content_string;
            return result;
        }
    }
    cout << "Not find the access."<<endl;
    return "";
}

bool access_bool(rec source, string name) {
    bool result;
    for (int i=0; i<source.length; i++) {
        if (source.r[i].name.compare(name) == 0) {
```

```
            result = source.r[i].content_bool;
            return result;
        }
    }
    cout << "Not find the access."<<endl;
    return 0;
}*/

fld access(tbl source, string fname) {
    for (int i=0; i<source.row_length; i++) {
        if (source.t[i].name.compare(fname) == 0) {
            if (source.t[i].type == 0) {
                vector<int> array;
                for (int j=0; j<source.col_length; j++) {
                    array.push_back(source.t[i].f_int[j]);
                }
                fld target = fld(&array[0], fname, source.col_length);
                return target;
            } else if (source.t[i].type == 1) {
                vector<double> array;
                for (int j=0; j<source.col_length; j++) {
                    array.push_back(source.t[i].f_double[j]);
                }
                fld target = fld(&array[0], fname, source.col_length);
                return target;
            } else if (source.t[i].type == 2) {
                vector<string> array;
                for (int j=0; j<source.col_length; j++) {
                    array.push_back(source.t[i].f_string[j]);
                }
                fld target = fld(&array[0], fname, source.col_length);
                return target;
            } else if (source.t[i].type == 3) {
                bool array[source.col_length];
                for (int j=0; j<source.col_length; j++) {
                    array[j] = source.t[i].f_bool[j];
                }
                fld target = fld(array, fname, source.col_length);
                return target;
            }
        }
    }
    cout << "Not find the access."<<endl;
    vector<int> null;
    fld empty = fld(&null[0], fname, 0);
    return empty;
}

rec access(tbl source, int num) {
    vector<tuple> tuples;
    if (num > source.col_length-1) {
        cout << "Exceed the scope."<<endl;
        rec target = rec(&tuples[0], 0);
        return target;
    } else {
        for (int i=0; i<source.row_length; i++) {
            if (source.t[i].type == 0) {
```

```
                tuple temp = tuple(source.t[i].f_int[num], source.t[i].name);
                tuples.push_back(temp);
            } else if (source.t[i].type == 1) {
                tuple temp = tuple(source.t[i].f_double[num], source.t[i].name);
                tuples.push_back(temp);
            } else if (source.t[i].type == 2) {
                tuple temp = tuple(source.t[i].f_string[num], source.t[i].name);
                tuples.push_back(temp);
            } else if (source.t[i].type == 3) {
                tuple temp = tuple(source.t[i].f_bool[num], source.t[i].name);
                tuples.push_back(temp);
            }
        }
        rec target = rec(&tuples[0], source.row_length);
        return target;
    }
}

tbl access(tbl source, int start, int end) {
    if (start > source.col_length-1) {
        cout << "Exceed the scope."<<endl;
        vector<rec> null;
        tbl target = tbl(&null[0], 0, source.row_length);
        return target;
    }
    vector<rec> store;
    for (int num = start; num < end; num++) {
        vector<tuple> tuples;
        for (int i=0; i<source.row_length; i++) {
            if (source.t[i].type == 0) {
                tuple temp = tuple(source.t[i].f_int[num], source.t[i].name);
                tuples.push_back(temp);
            } else if (source.t[i].type == 1) {
                tuple temp = tuple(source.t[i].f_double[num], source.t[i].name);
                tuples.push_back(temp);
            } else if (source.t[i].type == 2) {
                tuple temp = tuple(source.t[i].f_string[num], source.t[i].name);
                tuples.push_back(temp);
            } else if (source.t[i].type == 3) {
                tuple temp = tuple(source.t[i].f_bool[num], source.t[i].name);
                tuples.push_back(temp);
            }
        }
        rec temp = rec(&tuples[0], source.row_length);
        store.push_back(temp);
        if (end == source.col_length - 1) {
            break;
        }
    }
    tbl target = tbl(&store[0], store.size(), source.row_length);
    return target;
}



tbl access(tbl source, string *names, int length) {
    vector<fld> store;
```

```
    for (int k=0; k<length; k++) {
        string fname = names[k];
        for (int i=0; i<source.row_length; i++) {
            if (source.t[i].name.compare(fname) == 0) {
                if (source.t[i].type == 0) {
                    vector<int> array;
                    for (int j=0; j<source.col_length; j++) {
                        array.push_back(source.t[i].f_int[j]);
                    }
                    fld temp = fld(&array[0], fname, source.col_length);
                    store.push_back(temp);
                } else if (source.t[i].type == 1) {
                    vector<double> array;
                    for (int j=0; j<source.col_length; j++) {
                        array.push_back(source.t[i].f_double[j]);
                    }
                    fld temp = fld(&array[0], fname, source.col_length);
                    store.push_back(temp);
                } else if (source.t[i].type == 2) {
                    vector<string> array;
                    for (int j=0; j<source.col_length; j++) {
                        array.push_back(source.t[i].f_string[j]);
                    }
                    fld temp = fld(&array[0], fname, source.col_length);
                    store.push_back(temp);
                } else if (source.t[i].type == 3) {
                    bool array[source.col_length];
                    for (int j=0; j<source.col_length; j++) {
                        array[j] = source.t[i].f_bool[j];
                    }
                    fld temp = fld(array, fname, source.col_length);
                    store.push_back(temp);
                }
            }
        }
    }
    tbl target = tbl(&store[0], source.col_length, store.size());
    return target;
}

ostream & operator << (ostream & sys, const tuple &in) {
    sys << in.name << endl;
    if (in.type == 0) {
        sys << in.content_int;
    } else if (in.type == 1) {
        sys << in.content_double;
    } else if (in.type == 2) {
        sys << in.content_string;
    } else if (in.type == 3) {
        if (in.content_bool == 0) {
            sys << "false";
        } else {
            sys << "true";
        }
    }
    sys << endl;
    return sys;
```

```
}

ostream & operator << (ostream &sys, const rec &in) {
        for (int i=0; i<in.length; i++) {
                sys << in.r[i].name << "\t";
        }
    sys << endl;
        for (int i=0; i<in.length; i++) {
        if (in.r[i].type == 0) {
            sys << in.r[i].content_int;
        } else if (in.r[i].type == 1) {
            sys << in.r[i].content_double;
        } else if (in.r[i].type == 2) {
            sys << in.r[i].content_string;
        } else if (in.r[i].type == 3) {
            if (in.r[i].content_bool == 0) {
                sys << "false";
            } else {
                sys << "true";
            }
        }
                sys << "\t";
        }
        sys << endl;
        return sys;
}

ostream & operator << (ostream &sys, const fld &in) {
        sys << in.name << "\t" << endl;
    if ( in.type == 0 ) {
            for (int i=0; i<in.length; i++)
                    sys << in.f_int[i] << "\t" << endl;
    } else if ( in.type == 1 ) {
        for (int i=0; i<in.length; i++)
            sys << in.f_double[i] << "\t" << endl;
    } else if ( in.type == 2 ) {
        for (int i=0; i<in.length; i++)
            sys << in.f_string[i] << "\t" << endl;
    } else if ( in.type == 3 ) {
        for (int i=0; i<in.length; i++) {
            if (in.f_bool[i] == 0) {
                sys << "false" << "\t" << endl;
            } else {
                sys << "true" << "\t" << endl;
            }
        }
    }
        return sys;
}

ostream & operator << (ostream &sys, const tbl &in) {
    for (int i=0; i<in.row_length; i++) {
                sys << in.t[i].name << "\t";
    }
    sys << endl;
    for (int i=0; i<in.col_length; i++) {
        for (int j=0; j<in.row_length; j++) {
```

```
            if ( in.t[j].type == 0 ) {
                sys << in.t[j].f_int[i];
            } else if ( in.t[j].type == 1 ) {
                sys << in.t[j].f_double[i];
            } else if ( in.t[j].type == 2 ) {
                sys << in.t[j].f_string[i];
            } else if ( in.t[j].type == 3 ) {
                if (in.t[j].f_bool[i] == 0) {
                    sys << "false";
                } else {
                    sys << "true";
                }
            }
            sys << "\t";
        }
        sys << endl;
    }
    return sys;
}

/*int main(int argc, char const *argv[]) {
    int a[] = {90,99,98};
    fld b = fld (a , "value", getArrayLen(a));
    cout << b.f_int[0] << endl;
}
    int a[] = {90,99,98};
    string g[]  = {"ab", "cd", "ef"};
    fld b = fld (a , "value", getArrayLen(a));
    // sample definition of fld
    fld h = fld (g, "word", getArrayLen(g));
    cout << b;
    cout << h;
    tuple e[] = {tuple (22, "age"), tuple ("Fan", "name")};
    rec f = rec (e, getArrayLen(e));
    // sample definition of rec
    cout << f;
    tuple j[] = {tuple ("James", "name"), tuple (20, "age")};
    rec k = rec (j, getArrayLen(j));
    tuple u[] = {tuple (21, "age"), tuple ("Min", "name")};
    rec v = rec (u, getArrayLen(u));
    cout << k;
    fld i[] = {b,h};
    rec l[] = {f,k,v};
    tbl t = tbl(i, i[0].length, getArrayLen(i));
    // sample definition of tbl (kind 1)
    cout << t;
    fld tt = access(t, "word");
    cout << tt;
    rec ttt = access(t, 2);
    cout << ttt;
    tbl sss = access(t, 1, 3);
    cout << sss;
    string ta[] = {"word", "age", "value"};
    tbl stt = access(t, ta, getArrayLen(ta));
    cout << stt;
    tbl s = tbl(l, getArrayLen(l), l[0].length);
    // sample definition of tbl (kind 2)
```

```
    cout << s;
    tbl n = append(s, b);
    // sample definition of binding (kind 1)
    cout << "result of binding1 is:" << endl;
    cout << n;
    tuple m[] = {tuple (21, "age"), tuple ("Micheal", "name")};
    tuple y[] = {tuple (48, "age"), tuple ("Edwards", "name")};
    rec w = rec (m, getArrayLen(m));
    rec p = rec (y, getArrayLen(m));
    rec q[] = {w,p};
    tbl x = tbl(q, getArrayLen(q), q[0].length);
    cout << x;
    tbl z = append(s, w);
    // sample definition of binding (kind 2)
    cout << "result of binding2 is:" << endl;
    cout << z;
    tbl st = append(s, t, true);
    // sample definition of binding (kind 3)
    cout << "result of binding3 is:" << endl;
    cout << st;
    tbl sx = append(s, x, false);
    // sample definition of binding (kind 4)
    cout << "result of binding4 is:" << endl;
    cout << sx;
    tbl ss = modify(sx, 0, 1, "Cesc");
    // sample definition of changing
    cout << "result of changing is:" << endl;
    cout << ss;
    tbl sx1 = converse(sx, 0, "string");
    cout << "result of conversion is:" << endl;
    cout << sx1;
    tbl sx2 = converse(sx1, 0, "int");
    cout << "result of conversion is:" << endl;
    cout << sx2;
    // sample of data type conversion
    return 0;
}*/
```

dave_core.dave

```
float min_value(fld a) {
        if (a.type == 0) {
                int min = get_int(a, 0);
                int i;
                int length = a.length;

                for (i = 1; i < length; i++) {
                        int current = get_int(a, 0);
                        if (current < min) {
                                min = current;
                        }
                }
                return float(min);
        }
```

```
        else if (a.type == 1) {
                float min = get_float(a, 0);
                int i;
                int length = (a.length);
                for (i = 1; i < length; i++) {
                        float current = get_float(a, i);
                        if (current < min) {
                                min = current;
                        }
                }
                return min;
        }
        return 0.0;
}

float max_value(fld a) {
        if (a.type == 0) {
                int max = get_int(a, 0);
                int length = (a.length);
                int i;
                for (i = 1; i < length; i++) {
                        int current = get_int(a, i);
                        if (max < current) {
                                max = current;
                        }
                }
                return float(max);
        }
        else if (a.type == 1) {
                float max = get_float(a, 0);
                int length = (a.length);
                int i;
                for (i = 1; i < length; i++) {
                        float current = get_float(a, i);
                        if (max < current) {
                                max = current;
                        }
                }
                return max;
        }
        return 0.0;
}

float mean_value(fld a) {
        int length = a.length;
        if (a.type == 0) {
                int mean = 0;
                int i;
                for (i = 0; i < length; i++) {
                        mean += get_int(a, i);
                }
                return float(mean)/float(length);
        }
        if (a.type == 1) {
                float mean = 0.0;
                int i;
                for (i = 0; i < length; i++) {
```

```
                        mean += get_float(a, i);
                }
                return mean/length;
        }
        return 0.0;
}
```

ezcompile.sh

```
echo "Enter the Filename of Your DAVE Source Code File:"
read filename
echo "Option 1: Compile Your DAVE Source Code to C++ Source Code"
echo "Option 2: Compile Your DAVE Source Code to Executable"
read -p "[1,2]?:" input
if [ "$input" == "1" ]; then
        ./dave -c <$filename
        if [ $? != 0 ]; then
                echo "Failed to Compile to C++ Source Code."
                exit 1
        fi
        echo "$filename.dave Has Been Compiled to C++ Source Code (dave.cc)."
fi
if [ "$input" == "2" ]; then
        ./dave -c <$filename
        if [ $? != 0 ]; then
                echo "Failed to Compile to Executable (Unable to Compile to C++ Source Code with DAVE
Compiler)."
                exit 1
        fi
        g++ -w dave.cc
        if [ $? != 0 ]; then
        echo "Failed to Compile to Executable (Unable to Compile to Executable with g++ Compiler)."
        exit 1
        fi
        echo "$1 Compiled to Executable (a.out)."
fi
if [ "$input" != "1" ] && [ "$input" != "2" ] ; then
        echo "Invaild Input."
        exit 1
fi
```

test.sh

```
TESTFILES="../test/*.dave"
suffix=".dave"
prefix="../test/"
run=0
success=0

# run Makefile
make all

Compare() { #compare the two files
```

```
        diff -bq "$1" "$2" && {
            (( success++ ))
            echo "PASS"
        } || {
            echo "FAILED: does not match expected output"
        }
}

TESTING GOOD CASES
echo "Testing good cases:"
for f in $TESTFILES
do
        (( run++ ))
        echo "####################"
        name=${f#$prefix}
        name=${name%$suffix}
        ideal="../test/ideal/$name.out" #set the expect output path
        rm -f "../test/$name.out" #remove previous output
        echo "Testing: $name"
        ./dave -c < "$f" 2>&1 && {
                g++ -w dave.cc -Isrc 2>&1 && {
                        ./a.out > "../test/$name.out"
                        Compare "../test/$name.out" "../test/ideal"
                } || {
                        cat "../test/$name.out"
                        echo "FAILED: did not compile"
                }
        } || {
                cat "../test/$name.out"
                echo "FAILED: did not compile"
        }
done

echo "SUMMARY"
echo "Number of tests run: $run"
echo "Number Good Cases Passed: $success"
echo "#######################################"


# TESTING BAD CASES
BADFILES="../test/bad/*.dave"
badrun=0;
fail=0;

echo "Testing bad cases:"
for f in $BADFILES
do
        (( badrun++ ))
        echo "####################"
        name=${f#$prefix}
        name=${name%$suffix}
        ideal="../test/ideal/$name.out" #set the expect output path
        rm -f "../test/$name.out" #remove previous output
        echo "Testing: $name"
        ./dave -c < "$f" 2>&1 && {
                g++ -w dave.cc -Isrc 2>&1 && {
                        echo "FAILED: bad case successfully compiled"
```

```
                } || {
                        cat "../test/$name.out"
                        echo "PASS"
                        (( fail++ ))
                }
        } || {
                cat "../test/$name.out"
                echo "PASS"
                (( fail++ ))
        }
done

echo "SUMMARY"
echo "Number of tests run: $badrun"
echo "Number of Bad Cases Passed: $fail"

echo "#####################################"
echo "RESULT:"
echo "Number of good tests run: $run"
echo "Number Good Cases Passed: $success"
echo "Number of bad tests run: $badrun"
echo "Number of Bad Cases Passed: $fail"
```