Project goal: The goal of this project is to create a compiler for the AFSTAT language, from high-level text down to compilable and executable C or C++ code.

1 Description of the AFSTAT Language

The AFSTAT language (short for *Ask For Stat*) is a *domain-specific language* (DSL) whose purpose is to make it easier for analysts and statisticians to query data in order to perform statistical calculations.

Its syntax is *declarative*, meaning that the user describes what they want to do, not how to do it.

An AFSTAT program accesses data stored in CSV files, and typically begins with one or more **source** directives that indicate to the program which CSV files to read, followed by one or more **schema** definitions describing the structure of the data. The program may then contain **compute** operations for derived columns, **filtering** operations, **calculations** of results involving several columns, and finally an **analysis** block specifying statistical operations to perform on the data.

Comments can be added in the source code using `//`.

1.1 Input File (source)

This directive tells the program the path to the CSV file to be read and assigns it a logical name. Syntax:

```
:source source_name "path/to/file.csv";
```

1.2 Schema Declaration (schema)

The **schema** section defines the logical structure of the data. It is crucial for the generated C/C++ code to know how to read and interpret the CSV columns. Syntax:

```
schema schema_name {
    column_name data_type;
    ...
}
```

Supported data types are:

- **integer**: for integer numbers (e.g., age, number of patients)
- **float**: for decimal numbers (e.g., weight, temperature)
- **string**: for text strings (e.g., name, gender)

Example:

```
schema patient_data {
    id integer;
    age integer;
    gender string;
    weight float;
    height float;
}
```

Each file must always have its own schema, associating a schema with a data source. Syntax:

```
associate schema_name with source_name;
```

1.3 Definition of Computed Columns (compute)

The language allows the creation of new columns based on calculations using existing columns. Syntax:

```
compute new_column_name = expression;
```

Supported mathematical operators in expressions are `+`, `-`, `*`, `/`. Expressions can include parenthesized sub-expressions and integer or float constants.

Example: To calculate BMI (Body Mass Index), use the formula weight / (height * height):

```
compute bmi = weight / (height * height);
```

1.4 Specification of Statistical Analyses (analyze)

This is where the user requests the desired statistical calculations and visualizations. Syntax:

```
analyze schema_name {
    statistical_operation column_name;
    ...
}
```

Supported statistical operations:

- **mean**: computes the mean
- **median**: computes the median
- **std_dev**: computes the standard deviation
- **min**: finds the minimum value

- **max**: finds the maximum value
- **histogram**: generates a frequency histogram (for categorical or binned data)
- **summary**: produces a complete statistical summary (min, max, mean, std_dev) for a numeric column

Example:

```
analyze patient_data {
    summary age;
    mean weight;
    histogram gender;
}
```

Figures 1, 2, and 3 provide examples of AFSTAT programs used for medical data analysis.

1.5 Joining Multiple Data Sources in AFSTAT

An AFSTAT program can also access multiple CSV files to perform data joins. Instead of using just one `source` directive, several can be used.

Each file must always have its own schema, using the `associate` instruction:

```
schema schema_name { ... }
associate schema_name with source_name;
```

A **join** operation is used to combine data. It merges rows from two sources based on a matching condition. Syntax:

```
join join_name = source1.column1 inner join source2.column2;
```

```
// Simple demographic data analysis
// This program reads a patient file and calculates basic
   ↪ statistics
// on their age and gender.

// Specification of the input file
source patients "patients_medinfo.csv";

// Definition of the data structure
schema patient_data {
    id integer;
    age integer;
    gender string;
}
```

```
associate patient_data with patients;

// Analyses to perform
analyze patient_data {
    mean age;
    histogram gender;
}

// Expected results with the given data files
// mean age: 51.5
// histogram gender:
// M: 15
// F: 15
```

Figure 1 – Example of source code in AFSTAT language

This creates a new virtual data source that contains the columns of both sources for the rows where `source1.column1` equals `source2.column2`.

Note that several join operations can be chained, and these joins can use the names of previous joins, allowing the creation of joins involving more than two files.

Figures 4 and 5 show two examples of programs using join operations.

1.6 Data Filtering

Filtering allows selecting a subset of data for analysis. The `filter` instruction is placed after the declarations of sources and joins to which it applies, and before the analysis section.

It applies to a data source (either an initial source or the result of a join or filter) and produces a new virtual data source. Its syntax is:

```
filter id = data_source where condition;
```

The condition is of the form:

```
expression comparison_operator expression
```

An expression may use references to the source's columns, the same arithmetic operators as for computed columns, integer or floating-point constants, and parentheses. Supported comparison operators are:

`==`, `!=`, `<`, `>`, `<=`, `>=`

Figures 6 and 7 show examples of programs that use the `filter` instruction.

2 Error Detection

Special attention should be given to detecting programming errors by displaying the most explicit messages possible. All such errors must be detected before the compilation of the generated C/C++ program, which itself must compile without errors.

The concept of floating-point promotion will be used when a comparison or calculation involves both integer and floating-point operands.

The errors that must be detected are:

- Definition of a schema inconsistent with its associated data source. The error message must explicitly list the names of columns that do not exist in the source, or columns with types incompatible between schema and source.
- Use of an undefined ID.
- Join between incompatible columns (different types).
- Expression or calculation using incompatible operands.
- Analyses on incompatible columns (e.g., computing the mean of a string column).

```
// BMI calculation and analysis
// This program computes BMI before performing its analysis

// Specification of the input file
source visits "visits_info.csv";

// Definition of the data structure
schema health_data {
    id integer;
    weight float;
    height float;
}

associate health_data with visits;

// Definition of a new column (bmi)
compute bmi = weight / (height * height);

// Analyses to perform
analyze health_data {
    summary weight; // Weight summary
    summary height; // Height summary
    summary bmi;    // BMI summary
```

```
}

// Expected results with the given data files
// summary weight:
// min: 65.4
// max: 95.0
// mean: 78.8
// std_dev: 8.1
// summary height:
// min: 1.60
// max: 1.81
// mean: 1.70
// std_dev: 0.06
// summary bmi:
// min: 21.7
// max: 33.3
// mean: 26.5
// std_dev: 3.0
```

Figure 2 – Example of source code in AFSTAT language

2.1 C/C++ Code Generation

The generated code must be in C or C++, and it must compile without errors using the `gcc` or `g++` compiler.

3 Practical and Technical Aspects

The compiler must be written in C using the `flex` and `bison` tools. You must therefore design the grammar of the AFSTAT language.

This work must be done in a team of four students and submitted on the indicated date to your instructors during class and on Moodle. A final demonstration of your compiler will take place during the last lab session. You must submit your project on Moodle as an archive containing:

- The complete project code, which must compile simply with the command `make`. The default name of the generated C/C++ main program must be `afstat.c` or `afstat.cpp`.
- A document detailing the capabilities of your compiler — that is, what it can and cannot do. Be honest: highlight the interesting points you want the grader to see, since they may not be able to explore every part of the code.
- A test set.

Your compiler must provide the following options:

- `-version` must display the names of the project members.
- `-tos` must display the symbol table.
- `-o <name>` must write the generated C/C++ output code into the file `name`.

```
// Analysis with unused columns
// Even if the CSV file contains many columns, the DSL allows
   ↪ declaring
// only those of interest, thus simplifying the process.
// CSV file: "id,name,age,city,weight,height,bmi"
// The user is only interested in age and height.

source med "medical_records_full.csv";

schema data_subset {
    age integer;
    height float;
}

associate data_subset with med;

analyze data_subset {
    mean age;
    std_dev height;
}

// Expected results with the given data files
// mean age: 51.5
// std_dev height: 0.061
```

Figure 3 – Example of source code in AFSTAT language

4 Important Recommendations

Writing a compiler is a significant project; therefore, it must be built incrementally, validating each step on a smaller subset of the language and gradually adding features or optimizations.

An extreme and counterproductive approach would be to try to write the entire compiler at once before any debugging. The result of such a process would most likely be unusable — that is, a compiler that does not work at all or remains highly buggy.

Therefore, we recommend first developing a *functional but limited* compiler, restricted to translating simple instructions. Starting from such a working version, it will be easier to extend the compiler by adding features, handling more complex instructions, or introducing control structures. Even if your compiler does not ultimately meet every objective, it will still be able to generate correct and working programs!

It is essential that the analysis results are displayed on screen, so that you can verify that your programs execute correctly and produce accurate results!

4.1 Example 4 – Source Code in AFSTAT using Join

```
// Definition of data sources
source patients "data/patients.csv";
source records "data/records.csv";

// Definition and association of schemas
schema patients_schema {
    id integer;
    name string;
}
associate patients_schema with patients;

schema records_schema {
    patient_id integer;
    cholesterol float;
}
associate records_schema with records;

// Join data based on the patient ID
join patient_cholesterol = patients.id inner join records.
   ↪ patient_id;

// Analysis on the combined data
analyze patient_cholesterol {
    summary cholesterol;
}

// Expected results with the given data files
// summary cholesterol:
// min: 75.6
// max: 270.6
// mean: 224.2
// std_dev: 25.1
```

Figure 4 – Example of source code in AFSTAT language using join

5 Grading Guidelines

- If your project does not compile, crashes immediately, or fails to generate any correct C/C++ code, the grade will be **zero (0)**. The evaluator is not expected to search for code fragments that might appear correct. Your compiler must run and perform something correct, even if minimal.
- If you are short on time, it is better to do fewer things but do them well. **Even if incomplete, your compiler must be able to generate compilable and executable C/C++ programs.**
- Develop test cases — they are part of your work and will be evaluated.
- Do things in order and focus on what is required. The evaluator will only assess additional work (e.g., code optimizations) if what was requested works correctly.
- A clean and modular code organization will be highly appreciated.
- If you decide to extend your language or compiler with new features (e.g., support for new joins or statistical capabilities), **do so only once the basic functionality is complete**. Integrate these features with examples in your documentation and tests.

5.1 Example 5 – Source Code in AFSTAT using Join

```
// Definition of sources
source patients "data/patients.csv";
source visits "data/visits.csv";

// Definition and association of schemas
schema patient_data {
    patient_id integer;
    age integer;
    gender string;
}
associate patient_data with patients;

schema visit_data {
    patient_id integer;
    visit_date string;
    weight float;
    height float;
```

```
}
associate visit_data with visits;

// Joining data to analyze weight and height
join patient_visits = patients.patient_id inner join visits.
   ↪ patient_id;

// Calculate BMI for each visit
compute bmi = weight / (height * height);

// Analysis on joined data
analyze patient_visits {
    mean bmi;
    histogram gender;
}

// Expected results with the given data files
// mean bmi: 26.5
// histogram gender:
// M: 15
// F: 15
```

Figure 5 – Example of source code in AFSTAT language using join

5.2 Example 6 – Source Code in AFSTAT using Filter

```
// This program selects only patients over 50 years old
// before analyzing their data
source patients_data "data/patients.csv";
source records_data "data/records.csv";

schema patient_schema {
    id integer;
    age integer;
    gender string;
}
associate patient_schema with patients_data;

schema record_schema {
    patient_id integer;
    cholesterol float;
}
associate record_schema with records_data;
```

```
// Joining data
join all_data = patients_data.id inner join records_data.
   ↪ patient_id;

// Filtering to keep only patients older than 50
filter old_patients = all_data where all_data.age > 50;

// Analysis on the filtered subset of data
analyze old_patients {
    mean cholesterol;
    histogram gender;
}

// Expected results with the given data files
// mean cholesterol: 236.4
// histogram gender:
// M: 9
// F: 9
```

Figure 6 – Example of source code in AFSTAT language using filter

5.3 Example 7 – Source Code in AFSTAT using Filter on Strings

```
// Filtering on a string column
source employees "data/employees.csv";

schema employee_schema {
    id integer;
    name string;
    department string;
    salary float;
}
associate employee_schema with employees;

// Filtering to keep only employees from the "R&D" department
filter r_and_d_employees = employees where employees.department ==
   ↪  "R&D";

// Analysis on salaries of this group
analyze r_and_d_employees {
    summary salary;
}
```

```
// Expected results with the given data files
// summary salary:
// min: 65000
// max: 75000
// mean: 70000
// std_dev: 3000
```

Figure 7 – Example of source code in AFSTAT language using filter