COMP3173 24F

Project Description

Table of Contents

[**Prologue** 1](#_Toc179443249)

[**Language Feature** 2](#_Toc179443250)

[**Examples** 6](#_Toc179443251)

[**Phase 1 – Lexical Analysis** 8](#_Toc179443252)

[**Phase 2 – Syntax Analysis** 9](#_Toc179443253)

[**Phase 3 – Semantic Analysis** 9](#_Toc179443254)

[**Output** 9](#_Toc179443255)

[**Bonus** 14](#_Toc179443256)

# **Prologue**

**Project Title: Set Algebra Calculator**

**Grouping:**

Two students are in a group, no need to be in the same section. Once grouping is completed, students are not allowed to change their groups. Please follow the instruction in the email from Lily on Oct. 8 and complete grouping on **AUTOLAB**. Grouping is accomplished only if both two students in a group confirm.

**Objective:**

In this project, you will develop create a small calculator for Set Algebra. This algebra is simplified, which only consists of integers, set of integers, predicates, integer operators, set operators, logical operators, and relational operators. This language is designed to be straightforward and simple, making it easier for you to implement. So, some features look wired. Please accept if you see any and consider them as intentional design choices. Before commencing work on the project, it is imperative that you thoroughly review this document. Note that is **long**.

**The input** will be a source code file, which may include zero or multiple variable declarations (integer or set) and one single calculation expression. The calculator will then analyze the source code which is formulated in in Set Algebra. The analyzer should detect and highlight any errors presented in the source program; if no errors are found, the calculator will proceed with evaluation.

The first part of this document explains the language’s features with examples, which is like a programming language user manual. By reading it, students will gain a comprehensive understanding of the language, enable them to write expressions in this language and to identify errors in source code. The subsequent part of the document outlines the instructions of implementation by different phases. Students are expected to complete the project by sequentially completing these 3 phases.

**Implementation Language:**

**C or Python**, you have your own choice. Startup codes and instructions for both languages are given in the package.

**Grading:**

* Submission – 5%
* Compilation – 5%
* Phase 1 – 40%
* Phase 2 – 30%
* Phase 3 typing – 10%
* Phase 3 evaluation – 10%
* Bonus 1 “simplify” without Distributive Law – 2%
* Bonus 2 “simplify” with Distributive Law – 2%
* Bonus 3 function or inequality order testing – 1%

**DDL:**

We do not put DDLs for each phase. The DDL for the entire project will be **Dec. 17 midnight**, the last day of classes. But, **do not wait until the last minute**. Please expect that the project will cause **40 human hours**. Students will submit their works to **AUTOLAB**. (More instructions will be given later.) Marking will be purely based on testcases by AUTOLAB.

**Testcases:**

More testcases will not be with this project description. Because students can check the correct outcome at [Set Algebra Playground](https://set-algebra.pages.dev/). **(Please try this!!!**) Students can also write their own testcases and understand the language better.

# **Language Feature**

The user manual of this language is listed.

**Keywords**:

* “let”: initiates a variable declaration.
* “be”: acts as the assignment operator in other programming language.
* “int”: specifies the variable is of **integer type** during its declaration.
* “set”: specifies the variable is of **set type** during its declaration.
* “show”: acts as the main function in other programming languages, which initialize a calculation.

**Data types**:

* **Integer**:
  + Basic data type
  + A single-digit integer is an arbitrary decimal number.
  + A multi-digit integer starts with a non-zero decimal number and followed by any sequence of arbitrary decimal numbers.
  + Users can only declare non-negative integers. Negative integers are constructed through subtraction operations.
* **Arithmetic expression**:
  + Constructed data type
  + Users are **not** allowed to **declare** an arithmetic expression. There is no specific data type keyword for arithmetic expressions in the language. This data type only exists in the compiler. You can also check the keywords. There is no data type keyword for arithmetic expressions.
  + An **atomic arithmetic expression** is either an integer constant or an integer variable.
  + A compound arithmetic expression consists of two arithmetic expressions connected by an arithmetic operator (addition “+”, subtraction “-”, multiplication “\*”). And parentheses are used to define substructures within expression. For example,

1 + 2 – 3 \* 4

is parsed as:

( 1 + 2 ) - ( 3 \* 4 )

* **Predicates**
  + Constructed data type
  + Same as arithmetic expression, predicates are **not** directly **declared** by users but are instead managed within the compiler.
  + An **atomic predicate** is a relational comparison, which can be of two types:
    - **Integer value comparison**: involving the comparison of two integers using relational operator less than (“<”), greater than (“>”), or equality (“=”); or
    - **Membership testing**: used to determine if an element is a member of a set using the membership operator “@”.
  + A **compound predicate** is formed by combining “smaller” predicates (either atomic or compound) using logical operators:
    - **Binary logical operators**: two (atomic or compound) predicates connected by a conjunction (“&”) or disjunction (“|”)
    - **Unary logical operators**: Negation (“!”) which precedes another predicate.

For example,

P & Q

and

! R

where P, Q, and R are predicates.

* + Parentheses are also used to define substructures in predicates. Parentheses are essential for defining the precedence and grouping of operations within predicates, ensuring the correct evaluation of complex expressions.
* **Bool**:
  + Basic data type
  + **Cannot** be declared by users
  + Has only two constants: “true” and “false”
  + A Boolean is produced by the evaluation on a predication without uninitialized variables. For example,

x > 5

is a predicate if variable x has not been initialized. But if x has been initialized as 3 previously, then the predicate becomes

3 > 5

and can be evaluated to be Boolean “false”. The behavior of “>” will be explained later in this document.

* **Set**:
  + Constructed data type
  + Can be declared by users
  + A set is defined in using this syntax within the language

{ x : P(x) }

where

* a set defnition is enclosed within curly braces “{ }”;
* “x” is a variable name called representative, whose scope is limited within **this** **set** definition;
* “:” is another punctuation separating the representative x from the rest part of the definition;
* “P(x)” is a predicate that applies to the variable x, serving as the characteristic function of the set, which performs a logical test on x. If P(x) evaluates to **true**, then x is an element of the set; otherwise, x is not in the set.
  + This project focuses solely on sets of integers. Other types of sets, such as sets of strings, pairs, or sets of sets are not included. This limitation is intentional, aiming to simplify the implementation process. Goliath is trying to make your life easy!
* **Void**:
  + Basic data type
  + **Cannot** be declared by users
  + For subexpressions without any type

**Identifier**: arbitrary strings of English letters **in lower case** and are not reserved by keywords.

**Operators**:

* **Arithmetic operators**:
  + “+”: integer addition, calculates the sum of two integers
  + “-”: integer subtraction, calculates the difference of two integers
  + “\*”: integer multiplication, calculates the product of two integers
  + Multiplication has the highest precedence. Addition and subtraction have equal precedence, which is lower than multiplication.
* **Relational operators (for integers)**:
  + “<” (Less Than): returns “true” if the left-hand side integer is **less than** the integer on the right-hand side
  + “>” (Greater Than): returns “true” if the integer on the left-hand side is **greater** than the integer on the right-hand side integer
  + “=”: (Equal), returns “true” if the integer on the left-hand side is **equal** to the integer on the right-hand side integer
* **Relational operator (membership)**:
  + “@”: This operator checks if the element on the left-hand side is a member of the set on the right-hand side. It returns “true” if the element is **in** set, otherwise it returns “false”.
* **Logical operators**:
  + conjunction “&”, disjunction “|”, and negation “!” are behaved as the following.

**Truth Tables for Logical Operators:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| & | true | false |  | | | true | false |  |  |  |  |
| true | true | false |  | true | true | true |  | x | true | false |
| false | false | false |  | false | true | false |  | !x | false | true |
| Conjunction table | | |  | Disjunction table | | |  | Negation table | | |

* + Negation has the highest **precedence**, then conjunction, and then disjunction.
* **Set operators**:
  + “I”: set intersection, calculates the **intersection** of two sets
  + “U”: set union, calculate the **union** of two sets
  + Intersection has a higher **precedence** than union.

**Sentences**:

* Each **source code** contains zero, one, or multiple **variable declaration(s)**; and **exactly one calculation expression**.
* Each **variable declaration** is in the following syntax

let T id be E .

where

* + T is a type name (either int or set),
  + id is a variable identifier,
  + E is an expression that assigns a value (a set definition is also a “value”, even it is not a number) of the specific type to the variable.
  + The period “.” Marks the end of the declaration.

For example,

let int a be 5 .

defines an integer a whose value is 5. And

let set s be { x : x = 5 } .

defines a set s which contains only one integer 5.

* A **calculation expression** starts with keyword “show” and followed by an **algebraic expression**, which can be an arithmetic expression, a Boolean expression (a predicate with all variables initialized), or a set algebra expression. A calculation expression is also ended by a full stop “.”. For example,
  + To calculates the union of sets S1 and S2: show S1 U S2 .
  + To calculates the sum of integers 1 and 2: show 1 + 2 .
  + To test if integer 3 in S1 or not: show 3 @ S1 .

**Output**: After the calculation, the program prints the outcome on the screen (type and value). (see the examples below). For set operators, the “show” statement does not simplify the characteristic function. For example,

show { x : x > 3 } U { x : x > 5 } .

will output

{ x : ( x > 3 ) | ( x > 5 ) }

rather than

{ x : x > 3 }

even though the two sets are equivalent. This requirement will make this project easy. The simplification of predicates is an advanced feature and will be a bonus and explained later.

# **Examples**

* Example 1:

It simply prints integer 3. So, the expected outcome is



* Example 2:

Example 2 declares an integer x, whose value is 3. And it does a simple calculation. So, the outcome is



* Example 3

This example simply tests if the integer x is in the set s. The outcome is



* Example 4

This example declares two sets and calculates the intersection. The answer is



* Example 5

Subset can also be constructed.

The set y is a subset of x.

* Example 6

Example 6 has a lexical error because the letter “T” is not in the alphabet. “This” is not a keyword, so it has to be an identifier. However, identifiers are strings of English letters in lower case. Capital letters are excluded.

* Example 7

In this example, every word will be tokenized as an identifier. So, there is no lexical error. But, its structure is incorrect.

* Example 8

This is an example of type error. Both x and y are integers, but the relational operator “@” needs the second operator to be a set.

* Example 9

“show” statement can also calculate Boolean expressions (predicates without uninitialized variables). For example,

can be evaluated as



# **Phase 1 – Lexical Analysis**

**Timeline:** Students can start working on Phase 1 after “Lecture 3 – Finite Automata”.

**Description:**

In the first phase, students need to implement a lexer for this project. The lexer reads the source code as a stream of characters, cuts them into lexemes, classifies lexemes as tokens, and decides attributes for some lexemes. The lexer also reports lexical error (spelling mistakes) if the source code contains any. Tokens are previously defined in the language user manual and summarized again as below. The tokens that contain exactly one lexeme each are

* Keywords: let, be, show, int, set
* Punctuations: . , (, ) , {, }, :
* Arithmetic operators: +, -, \*
* Relational operators: @, <, >
* Logic operators: &, |, !
* Set operators: U, I

And the tokens that contain infinite lexemes are

* Integer constant: num
* Variable names: id

The space symbol “ ” is a special character, which is ignored by the lexer but terminates other lexemes by force. For example,

* “let be” is recognized as two tokens “let” and “be”, because the space symbol terminates the lexeme “let”.
* “letbe” is recognized as one token “id”, because it is a string of lower English letters but not a keyword.
* “be123+a” is recognized as four tokens “be”, “num”, “+”, and “id”.

**Symbol table** also needs to be implemented in this phase. It holds all variable names (ids) declared by users. Each variable name has another two attributes:

* the type of the variable (either “int” or “set”);
* the value of the variable.

Note that the value for a set variable is not a number, but a set definition. For example “let set s be { x : x = 5 }”, the value of “s” is “{ x : x = 5 }”. But in Phase 1, a lexer cannot decide a value for a set.

The lexer is operated as the function call “next\_token()” by the parser. It reads a few characters from the input and returns the first recognized token.

* If the token is an integer constant num, the function also returns its value.
* If the token is a variable name id, the function stores this variable name in a symbol table and returns the variable location in the symbol table (for example a point in C language).
* For other tokens, the function only returns the token name.

# **Phase 2 – Syntax Analysis**

**Timeline:** Students can start working on Phase 2 after “Lecture 8 – SLR Parser”

**Description:**

This is the second phase of a compiler. Students are expected to implement an **SLR(1) parser** for this language. The language grammar is given in “SLR Grammar.txt”. And the SLR(1) Parsing table is also given in “SLR Parsing table.csv” (your program can read the file as a text document). The parser works **with a stack**, which contains the already **parsed configuration**. The parser also needs to maintain a **syntax tree** to present the grammatical structure of the source code. A substructure of the tree is created when the parser reduces its stack. Each node in the syntax tree has **two attributes**: **type** and **value**. The attributes are left uninitialized in Phase 2, but will be used in Phase 3 Semantic Analysis.

# **Phase 3 – Semantic Analysis**

**Timeline**: Students can start working on Phase 3 after “Lecture 12 – Type System Part 2”. In fact, Lecture 11 and 12 are examples of semantic analysis for type checking, which is rather easy. Students are encouraged to self-study them and start working on Phase 3 as early as possible.

**Description:**

Semantic analysis is the last phase for the front-end of a compiler, which is the last piece of this project. In this part, the semantic analyzer checks the types and evaluates the source code. The analyzer also reports type errors if there is any.

The semantic analyzer should be implemented as a function, which

* contains the semantic definitions for all syntax rules, and
* execute one specific semantic calculation each time when the parser reduces its stack using the corresponding grammar rule.

The result of semantic analysis is stored by the attributes of each structure in the stack and the parse tree of the parser. (Recall that each structure of a grammar is combined with two attributes: type and value in Phase 2.) Thus, your parser needs to be modified a little to interact with your semantic analyzer.

The type checking rules are already given in “Type Checker.docx”. Students can use them directly. But for value evaluation, students need to design it by themselves.

# **Output**

The output of your compiler consists of the followings.

1. **A message on the screen** by standard output

If the source code has an error (lexical, syntax, or type), your compiler is supposed to identify it, show the error message (**“Lexical Error!”**, **“Syntax Error!”** , or **“Type Error!”**  respectively), and halts the analyzing procedure. This project has **no error recovering**. Furthermore, if any error is detected, the following json files are left as **empty** (created but empty).

If the source code has no error, your compiler needs to show a message (**“Lexical Analysis Complete!”**, **“Syntactic Analysis Complete!”**, or **“Semantic Analysis Complete!”**  respectively) to indicate the phases that have been implemented.

Furthermore, if you have implemented Phase 3 evaluation, your compiler also needs to **print the outcome of evaluation** on the screen.

1. **A file** named “**lexer\_out.json**” by file write

This file contains the outcome of the lexer (**lexemes and** the corresponding **tokens**) as **a list of dictionaries**. A list of objects is enclosed by a pair of square brackets “[ ]”. And a dictionary consists of two pairs and enclosed by a pair of curly brackets “{ }”. The first pair is a string “token” and the token name. The second pair is a string “lexeme” and the lexeme. For example,

“lexer\_out.json” should be



Note that spaces, indentations, and line breaks are not required. They are only used to improve the readability. Thus, the above example can also be



1. **A file** named “**parser\_out.json**” by file write

The outcome of Phase 2 must be written into “**syntax\_out.json**” using parentheses to present the hierarchical structure of the parse tree. Each non-terminal in the parse tree is a dictionary of two pairs. The first pair presents the name of the non-terminal. The second pair presents a list of children. Each terminal is presented in the same way as the outcome of the lexer. For example,



“syntax\_out.json” should be

Again, spaces, indentations, and line breaks are for readability only.

1. A file named “**typing\_out.json**” by file write

This file contains the type-checked parse tree in the same structure as “syntax\_out.json”. But each node in the tree contains one more attribute named “**type**”. For terminals, the order of attributes is “token”, “lexeme”, and “type”. For non-terminals, the order is “name”, “type”, and “children”. For the above example, “typing\_out.json” will be



1. A file named “**evaluation\_out.json**” by file write

Similar to “typing\_out.json”, this file holds the outcome of evaluation. So, each parse tree node has the attribute “evaluation” (“type” is omitted).

Note: The language is nicely designed. The grammar is an SLR(1) grammar and the semantic is L-attributed. Thus, these three phases can work simultaneously. Once the lexer tokenize a lexeme, the parser can start parsing and the type checker can check the types at the same time. So, you need a temporary memory location to hold the intermediate results (lexemes, tokens, the parse tree, etc.). If no error is found throughout the entire source code, these intermediate results are written to json files. Remember, json files are left empty if any error is found.

# **Bonus**

“This bonus is like the chocolate bite at the bottom of an ice cream.”

-CGY

The “show” statement can only do some calculations but cannot simplify a predicate in a set definition. For example,



simply returns



However, “0=0” is a tautology (always true). Thus, this set in fact is same as the set of all integers . Furthermore, a set can have multiple forms of definitions. For example,



also defines but is recognized differently as the previous example.

To overcome this issue, our language is extended with a new keyword “simplify”, which can simplify the predicate in a set definition under the following rules.

* An empty set is always presented as { a : false } because “false & P” is same as “false” by domination law. For example,

returns



* The universal set is always presented as { a : true }, similar to empty set. For example,

returns



* Predicates should be simplified into Disjunctive Normal Form (DNF aka sum of product). In general, a predicate in DNF consists of a bunch minterms and connected by disjunctions. And each minterm consists a bunch of atomic predicates (with or without negations) and connected by conjunctions. For example,

a = 1

( a = 1 ) | ( a = 2 )

( a > 1 ) & ( a < 3 )

( ( a > 1 ) & ( a < 3 ) ) | ( ! a > 5 )

are in DNF, but

( a = 1 ) & ( a = 2 | a =3 )

! ( ( a > 1 ) & ( a < 3 ) )

are not in DNF. After using DNF, all parentheses can be removed without any difficulty because of the precedence. The above DNF expression can be

a = 1

a = 1 | a = 2

a > 1 & a < 3

a > 1 & a < 3 | ! a > 5

In our language, each minterm is either

* + - a single larger or smaller comparison (for example “a > 0”, or “a < 3”) or
    - the conjunction of a larger or a smaller (for example “a > 5 & a < 10”). And the larger comparison always goes before the smaller comparison.
    - Furthermore, minterms need to be sorted in the increasing order by the lower bound. For example,

a > 10 & a < 15 | a < 3 | a > 5 & a < 8

is rearranged as

a < 3 | a > 5 & a < 8 | a > 10 & a < 15

because the lower bound of “a < 3”, “a > 5 & a < 8”, and “a > 10 & a < 15” are negative infinite, 5, and 10 respectively.

* + - And the range defined by two minterms do not intersect. For example,

a > 10 & a < 15 | a < 3 & a > 8

should be simplified as

a > 3 & a < 15

To accomplish these features, some evaluation rules need to be defined.

* + - “a = b” is evaluated as “a > b – 1 & a < b + 1”.
    - “a > b1 & a < b2 | a > b3 & a < b4” is evaluated as
      * “a > min(b1,b3) & a < max(b2,b4)” if b1b3b2b4 or b1b3b4b2;
      * “a > b1 & a < b2 | a > b3 & a < b4” otherwise.
    - If “x” is an initialized variable, it is evaluated as “x.value”. For example, “a>x” becomes “a > x.value”.
    - If “x” is an initialized set (which is in the form {a:P(a)}), the predicate “a @ x” is evaluated as “P(a)”. For example,

let set s be { a : a > 10 } .

let set t be { a : a > 5 | a @ s } .

the set t is evaluated as { a : a > 5 | a > 10 } and will be further evaluated as { a : a > 5 }.

* Note that Distribution Law (distribute conjunction to disjunctions) will be needed in some cases to simplify expressions. For example, the set intersection

{ a : a > 5 } I {a : a < 10 | a > 15 }

is same as

{ a : ( a > 5 ) & ( a < 10 | a > 15 ) }

but not in DNF. So, it can be simplified as

{ a : a > 5 & a < 10 | a > 5 & a > 15 }

and further simplified as

{ a : a > 5 & a < 10 | a > 15 }

* Even with the above implementation, “simplify” cannot work properly in some cases. An arithmetic expression may have multiplications, which can be in the predicate in a set definition. For example, a set can be

{ a : a \* a \* a + 2 \* a \* a + 5 < 10 }

To simplify the set definition, one must solve the cubic inequality, which is not easy already. Then, how about inequalities with higher orders?

Thus, our language excludes all functions or inequalities of order higher than 1. And your semantic analyzer needs to calculate order of functions. If the order is higher than 1, “simplify” reports a semantic error; otherwise, expressions can be simplified normally.

Bonus Grading

* 2% for “simplify” statement without Distribution Law
* 2% for applying Distribution Law
* 1% for function or inequality order testing (this feature will be tested separately)