Indian Institute of Technology Gandhinagar


---

CS 327

Language Documentation

---

Team Name: AZKABan

## Group Members

Narla Karthikeya (20110117)
Zeeshan (20110242)
Bhavini Korthi (20110039)
Chaudhari Ayush (20110042)
Arun Mani (20110023)

**Under the guidance of**

*Prof. Balagopal Komarath*
*Rohit Narayanan*

# About AZKABan

AZKABan is a programming language developed as part of the project component of the Compilers CS-327 course during Spring 2023. It is an easy to learn programming language. It follows a minimal yet efficient approach to programming. Its elegant syntax and fairly strictly typed nature makes it a suitable choice for developing programs with certain minimum functionalities. The source code for the compiler is hosted on a private repository on Github, https://github.com/Karthikeyanarla/CS327_Compiler_Project. It is built using a Tree Walk Interpreter. This Documentation introduces the reader to the concepts and features of the AZKABan language.

## CONTENTS

# 1) Lexical analysis:

## 1.1) Lines:

A physical line is a sequence of characters which terminate at an end-of-line sequence. Basically, the line of text in a file. Logical lines are meaningful expressions that can span across multiple physical lines. An AZKABan program is a set of such logical lines. A line that contains only spaces and tabs is ignored. The end of a program is marked by an EOF token. AZKABan is a very flexible language in terms of how a program can be broken down into physical lines. In fact, we could write entire meaningful programs in a single physical line with careful use of spaces between the logical lines. The use of Semicolon to determine separate logical lines will be mentioned in the following sections where we discuss the various language constructs.

## 1.2) Comments:

While writing programs, it is always a good practice to write down small notes/ comments. It is important that they do not interfere with the actual program itself. That is why we have a way to denote comments. Comments are enclosed within a pair of curly brackets {}. Comments are ignored by the syntax, hence they do not interfere with the actual code. It is important that you close every opened bracket, otherwise an error will be flagged

## 1.3) White spaces:

White spaces are necessary to separate the different tokens in a program. The white spaces (tabs, empty lines, etc.) are ignored everywhere in the program. Spaces at the beginning of each line are also ignored. Hence it is not mandatory that you provide indentation for different levels. But it might help with the ease of understanding.

# 2) Tokens:

## 2.1) Identifiers:

As the name suggests, it is used to uniquely identify a certain entity like a variable or a function. An identifier is a combination of alphabets (both lowercase and uppercase) and digits (from 0 to 9). The first character must not be a digit.
Case is significant ,i.e., AZKABan is different from azkaban.

## 2.2) Keywords:

Keywords are predefined words that have special meaning and define the syntax of the language. The following keywords are reserved and cannot be used as identifiers. BEGIN, END, IF, THEN, ELSE, TRUE, FALSE, WHILE, DO, PRINT, FOR, RETURN, BREAK.

## 2.3) Literals:

Literals are constant values which are possible under certain built-in types.

### 2.3.a) StringLiteral:

These are a sequence of characters defined strictly within double quotes, eg: ”azkaban”, “compiler”, etc.

### 2.3.b) NumLiteral:

They are implemented as fraction types. Eg: 3/5, 2/1, etc. Note that defining a variable as i:=3 will automatically take the form of 3/1. To initialise i as 3/5 just write i:=3/5

### 2.3.c) IntLiteral:

These are integers as per the mathematical definition. Examples: 1, 7, 100. To distinguish them from NumLiterals, we provide a tilde(~) while defining For eg: i:=~2. Negative integers are initialised as follows i:= -~2. There can or cannot be space between the - sign and the tilde, but there should not be any space between the tilde and the digit that follows.

### 2.3.d) FloatLiteral:

They are decimal values. Eg: 3.14, 5.61. To initialise a float variable with value 3, you will have to initialise it as, say i:=3.0 as by default 3 refers to a fraction in AZKABan.

### 2.3.e) BoolLiteral:

These correspond to the truth values and hence can hold only two values: TRUE and FALSE.

## 2.4) Operators:

We can perform various operations with the data that we have. We can broadly classify them as arithmetic, logical and relational operations. The symbols used to denote these operations are called operators. All these operators can be only used with certain types of operands. Any attempt to use incompatible operands will result in an error message. (number types include floats, ints and fractions).

### Arithmetic operators:

Addition(+)

Subtraction(-)

Multiplication(*)

Division(/)

Integer Division(//)

Modulo(%)

Power(@)

All these follow the regular associativity and precedence order.

Integer division refers to a quotient, i.e., 37 // 5 is equal to 5.

The operands for binary addition can be both numbers or both strings. You cannot add a string to a number.

For the other operators the two operands have to be number types.

Note: The + and - symbols can be used as both binary and unary operators.

We can use any number of unary operators, i.e. 1 - + - - - 2 will result in 3.

### Relational operators:

GreaterThan(>)

LessThan(<)

GreaterThanOrEqual(>=)

LessThanOrEqual(<=)

EqualTo(=)

NotEqualTo(<>)

Note that assign operation is done using ‘:=’ while the condition whether two quantities are equal or not is denoted by ‘=’.

Operands on either side can be any number type. If at least one side is a string, the other side must also be a string. If either side is a Boolean, then it will result in type error.

## Logical Operators:

Or(||)

And(&&)

These are the regular logical operators. The operands on either side of these should be Booleans.

## Unary Boolifying operator(^):

In certain languages with no Boolean types like Perl, there is an explicit Unary Boolifying operator. After applying the Boolifying Operator, in case of numbers all non zero numbers are True and 0 is False while for strings non empty strings are considered as True and the empty string is False. Even though we have Boolean type, we also provide the Unary Boolifying Operator.

Examples: PRINT(^5); prints True on the output screen.

a:=””{a is empty string}; b:=^a; PRINT(b); prints False on the output screen

## NOT operator:

This is a unary operator that negates the operand(should be boolean).

Eg: a:=TRUE; PRINT(!a);

## Augmented Assign Operator:

+=  -=  *=  /=  @=

An example of how to use these- a:=1; a+=2; PRINT(a); This shows 3 on the screen. With what value we initialise the variable, we have to use the same type of Literal as the operand to Augmented assign operator, i.e, an attempt to perform a+=2.0; would have resulted in a type error even if it is still a number type. Even the number type has to match.

## 2.5) Delimiters:

( ) [ ] ; ,

The above punctuations act as delimiters in the grammar.

2.5.1) (): A pair of round brackets is used to enclose expressions. The uses of round braces are given below:

To define the precedence order of operations. Eg- a := (2+3)*5

To specify the parameters of a function during a function definition.

Eg- FUNCTION add(a,b)

To call a function

Eg- PRINT(“hello world”); add(2,3);

2.5.2) []: A pair of square brackets are used while dealing with lists in the following ways (The operations and statements are explained later, now only mentioning the use of square braces):

Initialisation- T := LIST:INTEGER[~1,~3,~5];

Slicing- a := T[0]; b:=[1:2];

2.5.3) Semicolon (;): It is used to mark the end of a logical line. Eg- a:=3; PRINT(“hello”); It is also used inside the for construct to separate between initialisation,ending condition and updation. Eg- FOR i:=1; i<=10; i+=1.

2.5.4) Comma (,): It is used to separate the values in a list, parameters in a function definition and a function call.

# 3) Data Model:

## 3.1) Objects:

Objects are an abstraction for data. Every object has a type and a value. Depending on whether the value of the object changes, it can be classified as mutable or immutable. The various data types in our language are given below.

## 3.2) Standard types:

The built-in data types are given below:

### 3.2.a) NumLiteral:

These are implemented as fractions of the form a/b. A sign is shown if it is negative only.

### 3.2.b) FloatLiteral:

This is a number type that can hold decimal values.

### 3.2.c) IntLiteral:

These can only hold integer values.

### 3.2.d) BoolLiteral:

These can only hold the truth values True and False.

### 3.2.e) StringLiteral:

These can hold a sequence of characters. They are enclosed within double quotation marks(”).

### 3.2.f) Lists:

These are a collection of other objects. We have 3 types of lists. INTEGER, STRING, FLOAT, FRACTION and NONE lists that hold only integers, only strings and any data types respectively. Any attempt to add other data types in INTEGER and STRING lists will throw a static type check error message. An example of how an integer list is defined.

T := LIST:INTEGER[~1,~3,~4, ~2, ~7];

We can perform various operations like append, pop, etc. which will be discussed later.

# 4) Execution Model:

## 4.1) Structure:

The lines constituting a program are separated into blocks. Each block is an atomic unit and executed as such.

## 4.2) Naming and Binding:

Each name refers to an object. The data type of identifiers are not declared explicitly. They are identified by the value that the identifier refers to.

## 4.3) Scoping:

Scope defines the visibility of a name. A name is resolved using the nearest enclosing scope. Set of all scopes visible to a block is known as its environment. The scope extends to any block within the block in which it is defined. A block is enclosed within BEGIN and END tokens.

## 4.4) Exception:

When an invalid token is generated by the lexer it stops the program and throws an error message.

# 5) Expressions:

## 5.1) Arithmetic:

### 5.1.1) Unary operations:

The BNF grammar notation for a Unary expression is given below:

expr ::= expr | (-)expr | (+)expr

### 5.1.2) Binary operations:

The common binary operations that are addition, subtraction, division and multiplication are implemented in AZKABan. In order to account for precedence and use of parentheses, the grammar including unary operations is defined as follows:

exponential ::= precedence3 @ precedence3 | precedence3

precedence3 ::= integer | ( precedence1 ) | identifier | (+/-) precedence3

precedence2 ::= precedence3 | precedence3 (* or / or %) precedence3

precedence1 ::= precedence2 | precedence2 (+/-) precedence2

## 5.2) Assignment expressions:

The identifiers are assigned values using ‘:=’, for example i := 1 implies the NumLiteral 1 is assigned to i.

## 5.3) Comparison expressions:

The following comparison operators are valid:

> < >= <= = <>

## 5.4) Concatenation:

The operator + performs different operations on different data types. While it performs arithmetic addition on integers, floats and fractions, it performs concatenation on strings. Note that attempting to perform + between a number type and string will show a static type checking error.

## 5.5) I/O operations:

The keyword PRINT is used to display values on screen. Eg: PRINT (2+3);

The value that you need to print on screen is mentioned with a pair of brackets. As it is a logical statement of its own, we delimit it using a Semicolon(;). The semicolon serves the purpose of an end-of-line.

## 5.6) List Operations:

5.6.1) APPEND(a,b); - This method is used to add more elements to a list. a refers to the list to which we are adding b to the tail end.

5.6.2) POP(a); - This method removes the tail element of the list.

5.6.3) LEN(a); - This method returns the number of elements in the list.

5.6.4) ISEMPTY(a); - This returns the truth value regarding whether the list is empty or not.

5.6.5) HEAD(a); - Returns the head or the first element in the list.

5.6.6) TAIL(a); - Returns the tail of the list, i.e, the list except for its head.

## 5.7) Slicing:

Slicing is allowed in both strings and lists. We can obtain the ith value in a string or list, say A, by the expression A[i] To obtain the value of ith to jth values use the expression A[i:j+1] The indexing starts from 0. If i and j are not number types, the type checker will raise an error.

Couple of things to note:

1. If the length of the string is 5, A[0:7] will return A itself.
2. A[-2] returns the second last element in the list.
3. A[-3:4] will return the third last element. Once it records a negative value after the opening square braces, it avoids the number after colon.
4. A[2:-1] will return the elements from index 2 to the last element.
5. If i or j are floats or fractions, they are converted to the greatest integer less than the value, i.e, A[2.9:4.2] is equivalent to A[2:4].

# 6) Control Flow Tools:

## 6.1) IF statement:

If statements are used to check conditions to determine flow. If else statements follow the below structure:

IF (condition) THEN [statement list_1] ELSE [statement list_2] END

Eg: IF a=1 THEN
    PRINT (“ON”);
ELSE
    PRINT (“OFF”);
END

IF, THEN, ELSE and END are mandatory. There is no Simple If construct. Even if the else part is empty add ELSE END after the statement list_1. There is also no elif keyword either. This does not limit the ‘power’ of the code even though it increases the time taken for programs.

Also note that if statement list_2 is a set of more than one statements instead of a single statement, then obviously, we have to write BEGIN and END to enclose the statement list. In this case do not forget to add the END that is already a part of the IF-ELSE construct.

## 6.2) WHILE statement:

While is used to execute the same set of statements until the condition is no longer true. It follows the below structure:

WHILE (condition) DO [statement list] END

Eg: WHILE i<10
DO
    BEGIN
        j := j * i ;
        i := i + 1;
    END

## 6.3) FOR statement:

For statements are used to iterate through a set of instructions. The purpose of a for loop is similar to that of a while loop, but it is more robust because of its structure.

The for keyword is followed by 3 expressions separated by 2 semicolons in between. It follows the below structure:

FOR expression1 ; expression2; expression3 DO [statement list] END

expression1 is the initialization of the iterating variable, expression2 is the condition at which the loop stops and expression3 is the rule for updating the iterating variable.

Eg: q:=1;

```
FOR  i:= 1 ; i < 126 ; i := i + 1
DO
   BEGIN
         q:=q*i;
         PRINT (i);
         PRINT (q);
   END
END
```

## 6.4) BREAK:

A BREAK statement might be used within a FOR or WHILE loop. It terminates the nearest enclosing loop.

Eg: j := 12;

```
WHILE i<10
DO
   BEGIN
      j := j * i ;
      i := i + 1;
      IF j = 15
      THEN
      BREAK;
      ELSE END
   END
```

Once j becomes 15 even if i is less than 10, the loop will not proceed further.

## 6.5) CONTINUE:

A continue statement might be used within a FOR or WHILE loop. It skips whatever statements are left in that iteration. It executes the next iteration.

```
Eg:  j := 12;
     WHILE i<10
     DO
        BEGIN
           i := i + 1;
           IF i % 2 = 0
           THEN
           CONTINUE;
           ELSE END
           j := j * i;
        END
```

Here j is multiplied by only the odd values of i.

## 6.6) FUNCTIONS:

### 6.6.1) FUNCTION DEFINITION:

A function definition specifies a user-defined function object. The keyword FUNCTION introduces a function definition. Functions are not executed immediately after their definitions. They are executed once they are called upon. The execution of a function definition binds the name of the function in the current local scope to the function object. The functions are implemented as First class objects.

```
Eg: FUNCTION add(a,b)
        BEGIN
            RETURN a+b;
        END
```

Here add is the name of the function and a and b are the two parameters to the function.

### 6.6.2) FUNCTION CALL:

A function call executes the function binded to the name. To execute the add function defined above and print the value, We can write PRINT(add(1,2)); Please note that giving more arguments than accepted by the function definition does not show any error. Say the definition accepts n arguments, only the first n arguments passed are considered. If the arguments passed are less, then inside the function the additional arguments will not have been assigned any values. Hence it will show, Variable not defined error. Other example:

PRINT(add(add(1,2), add(2,3)));

As shown in the example we can pass functions as arguments to functions as well.

## 6.7) RETURN:

On executing the RETURN statement, we leave from the current function. As the name suggests the keyword RETURN is used to return a value to the function call.

Eg:
```
FUNCTION add(a,b)
    BEGIN
      RETURN a+b;
    END
```

X----------X