# Project Structure

- `bison.def.y` : Parser: Builds and executes the complete abstract syntax tree.
- `flex.def.y` : Lexer: Reads the file, tokenizes it, and passes the tokens to the parser.
- `src/` : Utility functions and structs:
    - `headers/` : Contains header files for utility functions.
    - `ast.c` : Provides functions to create AST nodes, print the tree, and retrieve names for AST node types.
    - `queue.c` : Implements a queue, including functions to enqueue elements and push parameters onto the stack.
    - `stack.c` : Implements a stack, including functions to push, pop, and search for variables.
    - `value.c` : Provides functions to create, copy, and free value pointers for the stack.

# Natural--

## Entry Point

The entry point of a program is a function called `main`, which serves as starting point for execution. It is defined as follows:

```
function main returns nothing() {
  STATEMENTS
}
```

Example:

```
function main returns nothing () {
    print(str(Hello, World!));
}
```

This function must always be present in a program, and it does not return any value.

## Data Types

There are three fundamental data types supported in this language:

- **i64**: Represents a 64-bit integer, suitable for numerical calculations.
- **string**: Represents a sequence of characters, enclosed within `str(...)` syntax.
- **array**: A collection of elements of the same type, which can either contain integer or string values.

Example:

```
num as i64 = b(X)42;
text as string = str(Example);
numbers as array;
numbers = create array of size b(X)5 as i64;
```

## Variables

The programming language supports statically and strongly typed variables, all of which are stored on the global stack. Variables must be declared with an explicit type before use. During declaration, a variable can be initialized and later reassigned.

Example:

```
 VAR_NAME as TYPE;
VAR_NAME as TYPE = VALUE;
VAR_NAME = NEW_VALUE;
```

# Number Representation

Numbers can be represented in various bases, using a special syntax:

```
 // Binary (Base 2)
bin as i64 = b(II)1101;
// Decimal (Base 10)
var as i64 = b(X)4;
// Hexadecimal (Base 16)
hex as i64 = b(XVI)a12b;
```

The notation `b(BASE)VALUE` specifies the numerical base using Roman numerals.

# Strings

Strings are sequences of characters, which can include escaped characters:

```
str as string = str(Test${0x0a});
```

Explanation:

- `str( ... )` marks the beginning of a string.
- `${ ... }` is an escape sequence representing any character. These are defined using the hexadecimal value of the desired character.

### Accessing Characters

Individual characters within a string can be accessed with the `.get` function, using their index:

```
STR_VAR.get(INDEX)
```

Example:

```
first_letter = message.get(0);
```

# Arrays

### Declaration

An array is created with a specific size and type:

```
VAR_NAME = create array of size SIZE as TYPE;
```

Example:

```
scores = create array of size 10 as i64;
```

### Adding Values

Values can be appended dynamically at the end of the array, but the size of an array cannot change:

```
VAR_NAME.push(VALUE)
```

Example:

```
scores.push(b(X)95);
```

Attempting to push beyond the defined size will result in an error.

### Accessing Elements

Elements within an array can be accessed with the `.get` function, using their index:

```
VAR_NAME.get(INDEX)
```

Example:

```
first_score = scores.get(b(X)0);
```

Attempting to access an index out of bounds will result in an error.

## Control Structures

The language supports essential control structures, including loops and conditional statements. Conditional execution is handled using `if` statements, which evaluate expressions to determine whether a block of code should execute. The `repeat` loop allows fixed or infinite iteration, which can optionally be controlled by conditions. These structures provide the foundation for decision-making and repeated execution, making the language suitable for a variety of programming tasks.

### Conditional Statements

To alter the control flow of the program, the `if` statement is used, allowing the user to execute different blocks of code based on conditions. Additionally, `else if` can be used to chain multiple conditions, enabling more complex decision-making.

```
if (CONDITION) then {
  STATEMENTS
} [else if (CONDITION_2) / else { STATEMENTS }]
```

Example:

```
if (power_level is greater than b(X)9000) then {
  print(str(It's over 9000${0x0a}));
} else {
  print(str(Not really impressive${0x0a}));
```

```
}
```

### Loops

Loops in this language allow for repeating a block of code a specified number of times or indefinitely, with optional conditions for execution. The `repeat` keyword initiates the loop, followed by a numeric constant or `infinite` for endless repetition. An optional condition can be included to control execution dynamically. The loop body is enclosed in curly braces `{}` and contains the statements to be executed. An alternative block using `or {}` can be defined for cases where the loop does not run at all.

```
repeat NUMBER / infinite times [as long as (CONDITION)] {
  STATEMENTS
} [or { STATEMEMENTS }]
```

Example:

```
repeat b(X)5 times {
  print(str(Hello, World!${0x0a));
}

false_val as i64 = b(X)0
repeat b(X)5 times as long as (false_val) {
  print(str(Hello, World!${0x0a}));
} or {
  print(str(Good bye!${0x0a}));
}
```

## Logical Operations

### Comparison Functions

To compare two values of the same type, one of the three following comparison functions can be used:

- `is equal to` : Checks if two values are the same.
- `is less than` : Checks if one value is smaller than another.
- `is greater than` : Checks if one value is larger than another.

Example:

```
if (score is greater than b(X)50) then {
  print(str(Passed));
}
```

### Logical Operators

If there are multiple conditions to be checked, the `and` and `or` operators can be used:

- `or` : Evaluates as true if at least one condition is met.
- `and` : Evaluates as true only if both conditions are met.

Example:

```
if (age is greater than 18 and has_id is equal to b(X)1) then {
  print(str(Allowed));
}
```

# Arithmetic Operations

The progamming language can use the following operations on integer values:

- `addition` : Adds together two number
- `subtraction` : Subtracts the second number from the first one.
- `multiplication` : Multiplies both numbers
- `division` : Divides the second number by the first one.

```
VAR_NAME = apply OPERATION_TYPE to (VAR_NAME/constant value and VAR_NAME/constant value
```

Example:

```
result = apply addition to (b(X)5 and b(X)10);
```

# Functions

This language supports functions, allowing users to define reusable blocks of code that can take parameters and return values. Functions enable reusability and improving code organization. They can be called with arguments, and their return values can be used in expressions or assigned to variables, making them a fundamental part of structured programming.

### Definition

Functions can be defined like this:

```
function FUNCTION_NAME returns TYPE/nothing (PARAMETER_LIST) {
  STATEMENTS
}
```

Example:

```
function add returns i64 (a as i64, b as i64) {
    return apply addition to (a and b);
}
```

### Execution

To execute a function use the `call` keyword like folloing:

```
call FUNCTION_NAME(ARGUMENTS);
```

Example:

```
sum = call add(10, 20);
```

# Built-in Functions

The programming language provides three built-in functions for handling user input and randomization:

- `VAR_NAME = random number up to NUMBER;` which assings a random number between 0 and `NUMBER`.
- `VAR_NAME = get number;` which retrives a numerical input from the user.
- `VAR_NAME = get line;` which reads a entire line of text input until new line character.