## Members


- Onur Şahin
  - ID: 21702236
  - Section: 2
- Adile Büşra Ünver
  - ID: 21703285
  - Section: 1
- Mehmet Alper Genç
  - ID: 21702645
  - Section: 2


# SAMARITAN

Samaritan is a programming language designed for IoT devices.

## Language grammar

```
<program> ::= <maybe_stmt_list> ;

<maybe_stmt_list> ::= |
                <stmt_list> ;

<stmt_list>:
        <stmt> |
        <stmt_list> <stmt> ;

<stmt> ::=
        <assignment> |
        <declaration> |
        <if> |
        <while> |
        <foreach> |
        <function_decl> |
        DO <stmt_list> END |
        RETURN <expr> SC |
        RETURN SC |
        <expr_stmt>
```

```
<if> ::=
        IF <expr> THEN <maybe_stmt_list> <elseif_list> <maybe_else> END

<elseif_list> ::=
        <elseif_list> <elseif>

<elseif> ::=
        ELSE_IF <expr> THEN <maybe_stmt_list>

<maybe_else> ::=
        ELSE <maybe_stmt_list>

<assignment> ::=
        <expr> EQ <expr> SC

<declaration> ::=
        VAR NAME COLON <type> EQ <expr> SC |
        CONST NAME COLON <type> EQ <expr> SC

<while> ::=
        WHILE <expr> DO <maybe_stmt_list> END

<foreach> ::=
        FOREACH NAME COLON <type> IN <expr> do <maybe_stmt_list> END

<expr_stmt> ::=
        <expr> ;

<function_decl> ::=
        FUN NAME LPR <maybe_arg_list> RPR COLON <type> BEGIN
          <maybe_stmt_list> END

<maybe_arg_list> ::=
        <empty> |
        <arg_list>

<empty> ::=

<arg_list> ::=
        NAME COLON <type> |
        NAME COLON <type> COMMA <arg_list>

<expr> ::= <expr> PLUS <expr> |
    <expr> MIN <expr> |
    <expr> MUL <expr> |
    <expr> DIV <expr> |
```

```
    <expr> GT <expr> |
    <expr> LT <expr> |
    <expr> GTEQ <expr> |
    <expr> LTEQ <expr> |
    <expr> EQEQ <expr> |
    <expr> NEQ <expr> |
    <expr> AND <expr> |
    <expr> ANDAND <expr> |
    <expr> OR <expr> |
    <expr> OROR <expr> |
    <expr> MOD <expr> |
    <term>

<term> ::=
        INT_LIT | NAME | STR_LIT | FLOAT_LIT |
        CHAR_LIT | TRUE | FALSE | <call> | <array> |
        <range> | <access> | <primitive_var> |
        <primitive_fn>

<call> ::=
        NAME LPR <maybe_expr_list> RPR

<array> ::=
        LBR <maybe_expr_list> RBR

<range> ::=
        <expr> DOUBLE_DOT <expr>

<primitive_fn> ::HUM_FN LPR RPR |
        AIRP_FN LPR RPR |
        AIRQ_FN LPR RPR |
        SOUND_FN LPR RPR |
        TIME_FN LPR RPR |
        CONNECT LPR <expr> RPR |
        PRINTLN LPR STR_LIT RPR |
        PRINTLN LPR STR_LIT COMMA <maybe_expr_list> RPR |
        READLINE LPR RPR

<primitive_var> ::=
        SWITCH

<access> ::=
        <expr> DOT SEND_INT LPR <expr> RPR |
        <expr> DOT RECV_INT LPR RPR |
        <expr> DOT NAME |
        <expr> LBR <expr> RBR
```

```
<maybe_expr_list> ::=
        <empty> |
        <expr_list>

<expr_list> ::=
        <expr> |
        <expr> COMMA <expr_list>

<type> ::=
        INT |
        DOUBLE |
        STR |
        VOID |
        CHAR |
        BOOL |
        CONNECTION |
        LBR <type> RBR
```

## Language Features

- Samaritan programs start with a function called `main` which has the type `fun(): Void` (a function that takes no argument and returns void).
- Singleline comment starts with `#` and ends with `#`. We don't support multiline comments. Because couldn't figure out how to count newlines in multiline comments.
- `switch` variable is a predefined array with size 10 that represents switches on the IoT device.

## Design decisions

- We decided that using keywords such as `then` and `end` are better than using braces ( `{}` ) because these can have different meanings for different language constructs and using keywords handles dangling else problem automatically.
- Instead of declaring variables as type first and name second our format is as follows: `var <name>: <Type>`. Because we believe that name of the variable should be more important than its type. Also this notation makes it easier to see declaration because every declaration starts with the same keyword.
- We thought primitive for loops are error prone. Instead we have `foreach` statement that iterates over the value. For ease of use, Samaritan users can define ranges so they can easily iterate over an interval of values.
- All statements that consist of multiple statements end with `end`. We thought that using

the same token for all increases reliability.

## Primitive Functions

### `air_pressure(): Double`

- returns air pressure
- Unit: Pascal

### `air_quality(): Double`

- returns air quality
- Unit: micrograms per cubic meter

### `sound_level(): Double`

- returns sound level
- Unit: Decibel

### `now(): Int`

- returns seconds passed from 1 January 1970
- Unit: Seconds

### `println('format string', arguments): Void`

`println` function is a special function that takes a format string and any number of arguments afterwards. In the format string, n-th `{}` is replaced by n-th argument after the format string. For example:

```
println("7 = {} + {}", 5, 2);
```

statement prints `"7 = 5 + 2"`

### `readLine(): Str`

Reads a line from standard input and returns it as `Str` .

# Language Construct

### `<program>`

Root construct of the program. Every SAMARITAN derives from this rule.

### `<stmt>`

A statement that represents all statements.

### `<assignment>`

Allows to assign value to a variable. For example `a = 5;` statement assigns 5 to variable `a`.

### `<declaration>`

Declaration of a variable. For example `var foo: Int = 2;` defines a variable named `foo` with type `Int`. A declaration can be done with either `var` or `const` keyword.

#### `<const>`

Represents const declaration. Variables that are defined using `const` keyword can not be mutated after declaration. Binding is done at compile time so while declaring a constant one can only use expressions that can be calculated in compile time such as literals.

### `<if>`

If branching. An if branch can be standalone, or chained with multiple `elseif` s and a single `else` block. Example:

```
if foo > 5 then
    println("foo is bigger than 5");
else
    println("foo is less than or equal to 5");
end
```

### `<while>`

While statement rule. Executes the body until the condition evalutes to `false`.

```
while now() < foo do
    println("not reached");
end
```

### <foreach>

If the expression is an array, it iterates over all values. If the expression is a range it enumerates all values between the range.

```
foreach i: Int in some_array do
    println("{}", i * i);   # prints squared #
end
```

### <expr_stmt>

An expression statement is an expression followed by a semicolon. Usually used for function calls.

### <function_decl>

A statement to declare a function. It starts with `fun` keyword and then the name of the function. Then it follows with the argument list enclosed by parantheses. After the argument list, the return type is specified. Finally, function body is defined. Functions can be declared in side functions.

```
fun foo(i: Int): Int begin
    # Func body... #
end
```

### DO <stmt_list> END

Compound block statements. Creates a new lexical scope between `DO` and `END` keywords.

```
var x: Int = 5;
do
    var x: Str = "inside block";
    println("{}", x); # inside block #
```

```
end
println("{}", x); # 5 #
```

### `RETURN <expr> SC` and `RETURN SC`

Returns from function. If it has no expression, then it returns `Void`.

### `<stmt_list>`

Any number of statements.

### `<expr>`

Represents all kinds of expressions.

### `<binary>`

A binary expression is an expression with two operands of type `<expr>` and an operator.

```
5 + 2; # 7 #
5 * 2 + 4; # 14 #
```

### `<unary>`

A unary expression is an expression with a operator in the front.

```
!true; # false
---5; # -5 #
```

### `<call>`

Represents a function call.

```
foo(arg1, arg2); # calling function foo #
```

### `<access>`

Represents accessing to a field.

```
foo.bar # bar field of object foo #
```

### `<array>`

Denotes an array literal. Starts with `[` token, then has values in the list and finishes with `]` token.

```
var arr: [Int] = [ 1, 2, 3, 4 ];
```

### `<range>`

Returns a range. Ranges can be used in `foreach` loops. Ranges are left inclusive and right exclusive. If the left hand side is bigger than right, the range iterates in reverse order.

```
0..8 # represents [0,8) #
```

## Data Types

### Void

`Void` type can have only a single value. Usually used for denoting functions that return nothing. Functions that have `Void` return type returns have implicit return statements at the end of the body.

### Int

Represents integer data type. Can hold all values in 32-bit two's complement notation.

### Double

Represents 64-bit floating point data type.

### Str

Represents a string of characters. A string can be concatenated by using `+` operator. Strings can be iterated in `foreach` statements. Type of each element in the iteration will be `Char`.

### Char

Represents single ASCII character.

### Bool

Booleans can hold either `true` or `false`. They are used in `if` and `while` statements.

### Connection

Represents an established connection to an address. Integers can be sent to the address using `sendInt()` method and received using `receiveInt()` method.

### Array types

An array type is denoted by brackets. For example an array of `Int` can be represented as `[Int]`. Array notation can also be nested. Therefore, `[[Int]]` is a two dimensional array of `Int`s.