
# CS 315
# Programming Languages

Project 1 Report

# Alt_HI

Bartu Atabek — 21602229 — Section 2
Utku Görkem Ertürk — 21502497 — Section 1
Ataberk Gözkaya — 21501928 — Section 1

# Table of Contents

## Introduction

In this report, our aim is to introduce the new programming language called “Alt_HI”. We will present the complete BNF description with the explanation of each language constructs and nontrivial tokens in our language. Finally, we will give an example program written in Alt_HI.

## Specifications

### 1. Complete BNF Grammar of Alt_HI

Complete BNF Grammar of Alt_HI is shown below. The language constructs and tokens will be explained after the BNF description.

**<program>** → <stmts>

```
          |<function_definitions><stmts>
          |<stmts><function_definitions>
          |<function_definitions>
          |<empty>

<function_definitions> → <function_definition>
                        |<function_definitions><function_definition>

<stmts> → <stmt>
          <stmt><stmts>

<stmt> → <conditional_stmt>
         |<expression_stmt>
         |<loop_stmt>
         |<function_call_stmt>
         |<variable_declaration>
         |<assignment_stmt>
         |<comment_stmt>

<non-if_stmt> → |<expression_stmt>
                |<loop_stmt>
                |<function_call_stmt>
                |<variable_declaration>
                |<assignment_stmt>
                |<comment_stmt>

<variable_declaration> → var <identifier>
                        |var <identifier><stmts>
```

**<comment_stmt>** → **#**<string>

**<function_call_stmt>** → <function_names> **<–** <function_arguments>
|<function_names>

**<function_names>** → **move**
|**turn**
|**grab**
|**release**
|**readData**
|**sendData**
|**print**
|<identifier>

**<function_arguments>** → <factor>
|<function_arguments> **,** <factor>
|<logical_value>
|<function_arguments> **,** <logical_value>

**<function_definition>** →func <chars> **<–** <function_parameters> **begin** <stmts> **end**
|func <chars> **begin** <stmts> **end**
|func <chars> **<–** <function_parameters> **begin** <stmts> **return** <expression_stmt> **end**
|func <chars> **begin** <stmts> **return** <expression_stmt> **end**

**<function_parameters>** → **var** <identifier>
|<function_parameters> **, var** <identifier>

**<identifier>** → <chars>

**<chars>** → <alphabetic_chars>
|<alphabetic_chars><alphanumeric_chars>

**<alphabetic_chars>** → <alphabetic_char>
|<alphabetic_chars><alphabetic_char>

**<alphanumeric_chars>** → <alphanumeric_char>
|<alphanumeric_chars><alphanumeric_char>

**<alphanumeric_char>** → <alphabetic_char>
|<digit>

**<digits>** → <digits><digit>
|<digit>

**<integer_number>** → <digits>
|**-**<digits>
|**+**<digits>

**<float_number>** → <digits>.<digits>
|**-**<digits>.<digits>
|**+**<digits>.<digits>

**<numbers>** → <float_number>
|<integer_number>

**<loop_stmt>** → <for_loop>
|<while_loop>

**<for_loop>** → **iterate from** <factor> **to** <factor> **begin** <stmts> **end**

**<while_loop>** → **iterate if** <conditional_expression> **begin** <stmts> **end**

**<expression_stmt>** → <conditional_expression>
|<numerical_expressions>

**<conditional_expression>** → <logical_value>
|<identifier>
|<conditional_expression_rules>
|<not_operator><conditional_expression_rules>
|<not_operator><identifier>
|<not_operator><logical_value>

**<conditional_expression_rules>** → <cond_expr_NTFs>
|<cond_expr_TFs>
|<conditional_expression_rules> <cond_expr_NTFs>
|<conditional_expression_rules> <cond_exp_TFs>

**<ident_number>** → <numerical_expressions> | <identifier>

**<cond_expr_NTF>** → <ident_number> <logic_opr_NTF> <ident_number>
|<ident_number> <logic_opr_B> <ident_number>

**<cond_exp_NTFs>** → <cond_expr_NTF>
|<cond_exp_NTFs><cond_expr_NTF>

**<ident_logic>** → <logical_value> | <identifier>

**<cond_expr_TF>** → <ident_logic> <logic_opr_TF> <ident_logic>
|<ident_logic> <logic_opr_B> <ident_logic>

**<cond_exp_TFs>** → <cond_expr_TF>
|<cond_exp_TFs><cond_expr_TF>

**<conditional_stmt>** → <matched_if_stmt>
|<unmatched_if_stmt>

**<matched_if_stmt>** → **if** <conditional_expression> <matched_if_stmt> **else begin** <matched_if_stmt> **end**
|<non-if_stmt>

**<unmatched_if_stmt>** → **if** <conditional_expression> **begin** <stmts> **end**
|**if** <conditional_expression> **begin** <matched_if_stmt> **end else begin** <unmatched_if_stmt> **end**

**<assignment_stmt>** → <identifier> **=** <numerical_expressions>
|<identifier> **=** <numbers>
|<identifier> **=** <conditional_expression>
|<identifier> **=** <function_call_stmt>

**<numerical_expressions>** → <numerical_expressions> + <divide_and_multiply>
|<numerical_expressions> **-** <divide_and_multiply>
|<divide_and_multiply>

**<divide_and_multiply>** → <divide_and_multiply> * <mod>
|<divide_and_multiply> / <mod>
|<mod>

**<mod>** → <mod> % <factor>
|<factor>

**<factor>** → <numbers>
|<identifier>
|(<numerical_expressions>)

**<newline>** → **\n**

**<alphabetic_char>** → A**...**Z
| a**...**z
| _
| $

**<digit>** → 0…9

**<logic_opr_TF>** → **&** | **|**

**<logic_opr_B>** → **~=** | **==**

**<logic_opr_NTF>** → **< | > | <= | >=**

**<not_operator>** → **~**

**<logical_value>** → **true|false**

**<string>** → combination of all ASCII character sequences

**<empty>** →

## 2. Language Constructs

**<program>:** This non-terminal is the abstraction of the main program. Since Alt_HI does not have a nested structure but have sequential function definitions and statements can be written and called in any order.

**<function_definitions>:** This non-terminal designates how the individual <function_definition> and more of them  can be put together.

**<stmts>:** This non-terminal designates how the individual <stmt> non-terminals can be put together. We can either put a single <stmt> or, we can add more <stmts> by adding them next to a <stmt> which increases writability and encourages writing line by line thus increasing readability as well. We can also put comments before, after or in between the statements.

**<stmt>:** This non-terminal is an abstraction of main kinds of statements we can write in our program such as conditional statements, expression statements, etc.

**<non_if_stmt>:** This non-terminal like the <stmt> non-terminal is an abstraction of main kinds of statements with the exception of conditional statements.

**<variable_declaration>:**  This non-terminal specifies what counts as a variable, which is defined by <identifier>. Type declaration in our language is implicit, just one type which is var, therefore there is no need to declare the different types(ex. int, String) of the variable when defining it. Furthermore, after the declaration of the variable, statements can be added.

**<comment_stmt>:** This non-terminal defines what counts as a comment, which is a <string> that could contain all the ASCII characters.<comment_stmt> can be used everywhere in the code.

**<function_call_stmt>:** This non-terminal denotes how to call a function. Functions can be called directly by specifying their name. Function arguments can be added when calling the function after the (“**<-“**) symbol.

**<function_names>:** This non-terminal defines what counts as a function name, which is used to differentiate between other function declarations/calls and other language constructs that uses characters to build such as variables. For increasing the readability and writability, primitive functions such as **move**, **turn**, **grab**, **release**, ... are reserved in order to be directly accessible. **Note(semantic meaning):** All primitive type of functions takes infinitive number of arguments or none. For, **move**, **turn**, **grab**, **release**, **sendData**, last argument of these functions will be treated as quantity and other arguments will be treated as sensor IDs. For, **readData and print** all arguments will be treated as sensorIDs. If there is a <logical_value> it will be treated as 1(**true**) and 0(**false**). If no argument will be given all primitive functions will be act as default. As an example, if move will be called sensor (motor) 1 will be activated to move forward 1mm.

**<function_arguments> :** This non-terminal denotes what are accepted as arguments to function calls. Function argument are defined by <factor> and <logical_value>, and can be used side by side separated by a comma (‘,’).

**<function_definition> :** This non-terminal specifies how a function can be declared by using the reserved word **func**. In Alt_HI functions can be defined in four different ways. A function could take parameters and have a return statement or it’s combinations according to its needs. Function parameters which are defined by <function_parameters> can be added to the function definition using the left arrow (‘<-‘). Statements can be declared in the function definition between the reserved words **begin** and **end.** Furthermore, a return statement can be added with the reserved word **return** before the **end** reserve word.

**<function_parameters> :** This non-terminal denotes what are accepted as parameters to function definitions. We can have functions that receives no parameters, a single parameter, or multiple parameters that are separated by a comma (‘,’).

**<identifier>:** This non-terminal what counts as an identifier, which is defined by <chars>

**<chars>:** This non-terminal defines what counts as alphabetic characters followed by alphanumeric characters.

**<alphabetic_chars>:** This non-terminal allows left recursive definitions of <alphabetic_char>.

**<alphanumeric_chars>:** This non-terminal allows left recursive definitions of <alphanumeric_char>.

**<alphanumeric_char>:** This non-terminal defines what counts as an alphanumeric character which could consist of an <alphabetic_char> or a <digit>.

**<digits>:** This non-terminal allows left recursive definitions of <digit>.

**<integer_number>:** This non-terminal defines what counts as an integer, which is all the numbers that are a single digit, or a digit that is continued with integers. Furthermore, integers could have positive and negative signs by adding the positive (‘+’) or negative (‘-‘) symbols before the number. The numbers without signs will be considered as positive integer number.

**<float_number>:** This non-terminal defines what counts as a float number, which have decimal points denoted by the decimal symbol (‘.’). Like integers, floating numbers could have positive and negative signs by adding the positive (‘+’) or negative (‘-‘) symbols before the number as well as The numbers without signs will be considered as positive float number.

**<numbers>:** This non-terminal is an abstraction for all the numbers (i.e. <integer_number> and <float_number> ) we can write in our program.

**<loop_stmt>:** This non-terminal is an abstraction that contains all types of loop statements we can write in Alt_HI.

**<for_loop>:** This non-terminal denotes how to write for loop statements, which are used to execute certain statements iteratively until the condition is satisfied. We use **iterate** and **from** reserved words followed by a starting <factor> followed by the reserved word **to** and an ending <factor> which is followed by statements within the reserved words **begin** and **end**.

**<while_loop>:** This non-terminal denotes how to write while loop statements, which are used to execute the enclosed statements continuously until the condition is unsatisfied. We use **iterate** and **if** reserved words followed by a <conditional_expression> followed by statements within the reserved words **begin** and **end**.

**<expression_stmt>:** This non-terminal is an abstraction that contains different types of expressions such as conditional expression and numerical expression.

**<conditional_expression>:** This non-terminal is an abstraction that contains conditional expressions. It is important to see that <not_operator> has lower precedence over other conditional operations.

**<conditional_expression_rules>:** These are the rules that combination of all TF and NTFs expressions. (Explanation of TF, NTF and B will be given in <logic_opr_…> variables)

**<ident_number>:** This is the abstraction of <numberical_expression> and <identifier>.

**<cond_expr_NTF>:** This non-terminal gives ability of write 1 **==** 3, or 1 **==** <identifier> such statements which gives <logical_value> at the end.

**<cond_expr_NTFs>:** This non-terminal left recursive definition of <cond_expr_NTF> or a single <cond_expr_NTF>.

**<ident_logic>:** This is the abstraction of <logical_value> and <identifier>.

**<cond_expr_TF>:** This non-terminal gives ability of write true & false , or true & <identifier> such statements which gives <logical_value> at the end.

**<cond_expr_TFs>:** This non-terminal left recursive definition of <cond_expr_TF> or a single <cond_expr_TF>.

**<conditional_stmt>:** This non-terminal is a the abstraction for the conditional statements.

**<matched_if_stmt>:** This non-terminal contains a conditional statement starting with the reserved word **if** followed by a <conditional_expression> and a recursive <matched_if_stmt> followed by the reserved word **else** and a <matched_if_stmt> between the reserved words **begin** and **end**.

**<unmatched_if_stmt>:** This non-terminal contains a conditional statement with only one **else**, or a recursive rule for creating additional conditional statements.

**<assignment_stmt>:** This non-terminal specifies the ways to bind a value at the RHS ( right hand side) of the assignment operator (‘**=**‘) to a variable at the LHS (left hand side) of the assignment operator. We can make a variable hold the evaluated value of a numerical expression, the return value of a function, assign a logical value of a conditional expression or assign a number.

**<numerical_expressions>:** This non-terminal is the abstraction for the arithmetic operations such as (‘%’), (‘*’), (‘/’), (‘+’) ,(‘-’) in terms of mathematical precedence.

**<divide_and_multiply>:** This non-terminal is the abstraction for the multiplication, division and, modulo arithmetic operations denoted with the following operators (‘*****’), (‘**/**’), (‘**%**’) respectively with mathematical precedence.

**<mod>:** This non-terminal specifies the modulo arithmetic operation denoted with the modulo operator (‘**%**’).

**<factor>:** This non-terminal is an abstraction containing <numbers>, <identifiers> and <numerical expressions> in parenthesis.

**<newline>:** This non-terminal denotes what counts as a new line with the terminal **\n**.

**<alphabetic_char>:** This non-terminal defines what counts as an alphabetic character which are all the letters from A to Z including the small case letters with the addition of underscore (‘_’) and the dollar sign (‘$’).

**<digit>:** This non-terminal defines what counts as a digit, which are all the numbers form 0 to 9.

**<logic_opr_TF>:** This non-terminal is the abstraction of boolean logical operators and operator (‘**&**’) and or operator (‘**|**’) .TF denotes that just used with logical values.

**<logic_opr_B>:** This non-terminal is the abstraction of equitable logical operators containing not equal (‘**~=**’) and is equal (‘**==**’) operators. B denotes that this terminals can be used with both logical and numerical process.

**<logic_opr_NFT>:** This non-terminal is the abstraction of comparative logical operators containing less than (‘**<**‘), less than or equal (‘**<=**’), greater than (‘**>**’) and, greater than or equal (‘**>=**’) operators.NFT denotes that this terminals can be used with only numerical processes.

**<not_operator>:** This non-terminal specifies the not (‘**~**’) operator in logical calculations.

**<logical_value>:** This non-terminal defines what counts as a logical value. We have two logical values; **true** and **false.**

**<string>:** This non-terminal defines what counts as a string which is any sequence of character/s i.e., letters, numerals, symbols and punctuation marks (ASCII characters).

**<empty> :** This non-terminal is used in the BNF description to make an empty source code.

## 3. Tokens of Alt_HI

### 3.1 Reserved Words

The reserved words of Alt_HI are var, move, turn, grab, release, readData, sendData, print, func, takes, begin, end, iterate, to, iterate, if and, else, true and false. We tried to make a programing language that can be understood and interact as a natural language. That’s why these words are selected to increase readability and writability. We do not use separators (such as ‘;’) to be more human understandable and writable. This makes language less understandable if the programer choose to use Alt_HI in one line. That’s why we are recommending Alt_HI programers to use newlines.

### 3.2 Comments

Comments can be created by adding the ‘#’ symbol in front of a string. The whole line will be considered as a comment.

### 3.3 Identifiers

In Alt_HI, identifiers, which we simply names that can hold values, are used for defining the variables and for referring them in various statements. Identifiers are constructed of alphanumeric characters with the exception of the first character being an alphabetic character. This convention is used for increasing readability.

### 3.4 Literals

Literals are strings and chars, strings combination of all ASCII characters sequences and they only used in comments. Chars are used for just for identifiers.

## Example Program written in Alt_HI

```
# variable definitions

var noOfTurns
var isBox1Grabbed = false
var isBox2Released = true

# this is an assignment
noOfTurns = 20
#this is a loop
iterate if noOfTurns > 0
begin
   #this is a function call
   move
   turn
   noOfTurns = noOfTurns - 1
end

#this is a function definition
func doTheThing <- var isItGrabbed
begin
   #read data from sensor 13 and move - 1
   var sensor13 = readData <- 13
   move <- sensor13 - 1.2
   #0 left, 1 right
   if readData <- 5 > 3 #examine
   begin
      turn <- 0, 24.6
   end
   release
   return true
end

isBox1Grabbed = doTheThing <- isBox1Grabbed
if isBox1Grabbed
begin
   pickBox2
   isBox2Released = false
end

func pickBox2
begin
   iterate from 1 to 10
   begin
      move
   end
   grab
end

var isOperationSum = 0
var isOperationSubtract = 1
```

```
var isOperationMultiply = 2
var isOperationDivision = 3
var isOperationModulus = 4
var fail = false

func calculate <- var operation, var no1, var no2
begin
   if operation == isOperationSum
   begin
      return no1 + no2
   end
   if operation == isOperationSubtract
   begin
      return no1 - no2
   end
   if operation == isOperationMultiply
   begin
      return no1 * no2
   end
   if operation == isOperationDivision
   begin
      return no1 / no2
   end
   if operation == isOperationModulus
   begin
      return no1 % no2
   end
   return fail
end

var result = calculate <- 0, 1, 2.5
#send sensor 6 to resulting number
sendData <- 6, result
```