**Green World Assembly Language (GWAL) Specification**

(V. Cavalli-Sforza, Draft of Dec 6, 2021)

1. **Introduction and General Description**

The Green World Assembly Language (GWAL) is the language that GWL gets translated to in your project. It defines a virtual machine whose language is interpreted.

* 1. **Program Structure**

GWAL programs are divided into a *Data Definition* Section, an *Executable Instructions* Section, and an optional *File Input* Section*.* The overall program structure is defined below:

*<Program> ::=* **PROGRAM** *<name> <newline> { <BlC> }***DATA**{ *<dataline> | <BlC>* }  
**EXECUTE**{ *<instrline>* }  
[ **FINPUT** *<inputval>* { *<inputval>* }]

*<BlC> ::=* ( *<blanks> | <comment>* ) *<newline>*

*<dataline> ::=* <*datadef> <newline>*

*<instrline> ::= <instr> <newline>*

*<name>* is a user-defined identifier; *<datdef>* is a data definition; *<instr>* is an instruction; *<inputval>* is an integer or string literal, *<comment>* is a single-line comment. These are specified below. *<blanks>* just refers to the space or tab characters.

A sample program is shown in Section 3.

* 1. **Line Format**

GWAL data definitions and instructions follow a fixed format that can be described simply as follows:

*<Line> ::= <Field1>* [*TAB < Field2> TAB < Field3> TAB < Field4> TAB < Field5>*]

In other words, declarations and instructions are written in a 5-column format, separated by the TAB character and terminated by a newline (with one exception—see below). Each column is called a *field.* One or more of the fields may be blank. Here is a general example:

Field1 Field2 Field3 Field4 Field5

Specific examples will be provided below.

* 1. **Lexical Description**

GWAL contains the following types of lexemes: alphanumeric identifiers, integer literals, string literals. The following restrictions apply:

* **Reserved words** include the words **PROGRAM**, **DATA**, **CODE**, **FINPUT**, reserved identifiers used in data definition, and all the instruction opcodes (see below).
* **User-defined identifiers** (**UDIs**): alphabetic characters are strictly uppercase; up to 9 characters long.   
  Syntax: **[A-Z][A-Z0-9]\***
* **Integer literals**: can optionally be preceded by a sign and begin with 0’s; up to 9 characters long.   
  Syntax: **(+|-)?[0-9]+**
* **String literals**: can contain any alphabetic, numeric, or punctuation characters. Up to 9 characters can fit in a field, otherwise they can be specified on a separate line. See the next section.
  + 1. **Data Definition Section**

The Data Definition Section consists of 0 or more lines with the following syntax:

*<DataDef> ::= <Var/Const> TAB <UDI> TAB <Type>* [*TAB <Size>* [*TAB <InitValue>*]]  
*<Var/Const/Lit> ::=* **VAR**, **CONST***<UDI> ::= user-defined ID according to the syntax above  
<Type> ::=* **INT**, **STRING***<Size> ::= a positive, nonzero integer literal  
<InitValue> ::- integer literal | string literal*

*<Var/Const>* indicates whether this location is modifiable (**VAR**), not modifiable (**CONST**).

*<Size>* is 1 by default. Values of more than 1, indicate that this is an array (of INT).   
<*InitValue>* is an optional initialization value. If unspecified, the data location is set to 0; if specified, size must be too (with one exception).

* For **INT**, if it is provided, it is used to initialize all elements of the array to that value.
* For **STRING** data:
  + If no size is provided, the value 9 will be used as the default.
  + If size 0 is provided, it indicates that the program should calculate the size from the initial value. If the initial value is not provided, the default value 9 will be used.
  + If the size of the initial value and the size provided are in conflict, the larger of the two will be used.
  + If the string literal exceeds the 9 characters allowed in each field, use ‘\*’ as the initial value to indicate that the string is shown on the following line, but it cannot exceed 49 characters.

Examples:

CONST ONE INT 1 1

VAR ACC INT 1 0

VAR TEMP1 INT 1

VAR TEMP2 INT

VAR ARRAY1 INT 10

VAR ARRAY2 INT 20 100

VAR STR1 STRING

CONST GREET1 STRING 5 Hello

CONST GREET2 STRING 0 Bye

CONST ERROR STRING 0 \*

A long string on this line: up to 47 characters!!

* + 1. **Executable Instructions Section**

GWAL instructions follow a fixed format that can be described simply as follows:

*<Line> ::= <Label> TAB <Operator> TAB <Operand1> TAB <Operand2> TAB <Operand3>*

The *<Label>* field may be empty, in the sense of containing just blanks (and a \t) in order to maintain the fixed column format. In fact, the *<Label>* field will most often be empty.

Other fields maybe empty as well, if the *<Operator>* (opcode)does not require all of the operands. An alternative to leaving it empty is to place the single character ‘\*’ in that field (but not in the *<Label>* field).

Examples:

LOOP1 ADD TEMP1 TEMP2 TEMP3

ASSIGN 0 ZERO

ASSIGN 0 ZERO \*

READ NEWVAL

READ NEWVAL \* \*

STOP

* 1. **Comments**

Single line comments are permitted and must start at the left margin (first column) with ##.

1. **Instruction Set**

The instruction set of GAWL strongly resembles that of the pseudo-code interpreter of Unit 3, but there are some significant differences, the most obvious of one is the use of symbolic operators and operands.

In the operation descriptions below:

* The symbol ‘\*’ is used below to indicate that the operand is not used.
* *literal* refers to a literal value occurring in the instruction itself (an integer literal or short string).   
  Note that if a string literal is used in an instruction, it must be double quoted (so max length 7 chars).
* *location* refers to a data memory location (variable or constant) as defined in the Data Definition Section of the program and is indicated by an identifier.
  1. **Basic Assignment Operations**

The following operators are defined with the given syntax:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **OPERATOR** | **OPERAND 1** | **OPERAND 2** | **OPERAND 3** |
| Simple assignment opd3 ⟵ opd1 | ASSIGN | *literal | location* | \* | *location* |
| Assign from array element opd3 ⟵ opd1[opd2] | RARRAY | *location* | *literal | location* | *location* |
| Assign to array element opd2[opd3] ⟵ opd1 | WARRAY | *literal | location* | *location* | *literal | location* |

* 1. **Arithmetic Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **OPERATOR** | **OPERAND 1** | **OPERAND 2** | **OPERAND 3** |
| Addition opd3 ⟵ opd1 + opd2 | ADD | *literal | location* | *literal | location* | *location* |
| Subtraction opd3 ⟵ opd1 - opd2 | SUB | *literal | location* | *literal | location* | *location* |
| Multiplication opd3 ⟵ opd1 × opd2 | MUL | *literal | location* | *literal | location* | *location* |
| Division (truncating) opd3 ⟵ opd1 / opd2 | DIV | *literal | location* | *literal | location* | *location* |

* 1. **Branching and Looping**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **OPERATOR** | **OPERAND 1** | **OPERAND 2** | **OPERAND 3** |
| Branch on equal | JEQL | *literal | location* | *literal | location* | *label* |
| Branch on less | JLESS | *literal | location* | *literal | location* | *label* |
| Branch on greater | JGRTR | *literal | location* | *literal | location* | *label* |
| Loop (inc, test, jump) | LOOP | *location* | *literal | location* | *label* |
| Jump (unconditional) | JUMP | \* | \* | *label* |

* 1. **Input / Output**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **OPERATOR** | **OPERAND 1** | **OPERAND 2** | **OPERAND 3** |
| Read from file (FINPUT) | READFILE | \* | \* | *location* |
| Read from user (keyboard) | READ | \* | \* | *location* |
| Write to screen | WRITE | *literal | location* | \* | \* |

* 1. **Stack Manipulation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **OPERATOR** | **OPERAND 1** | **OPERAND 2** | **OPERAND 3** |
| Push element on stack | PUSH | *literal | location* | \* | \* |
| Pop element from stack | POP | \* | \* | *location* |

Contrary to the simple pseudo-code we saw in Unit 3, the VM defined by GWAL supports subprogram call and return. The stack is a data area that is distinct from the data defined in the Data Definition Section (think of that as your global/static variables) and separately managed. You can only manipulate the stack via the PUSH and POP operations, which will be useful to you as you implement subprogram call and return.

* 1. **Stopping Execution**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **OPERATOR** | **OPERAND 1** | **OPERAND 2** | **OPERAND 3** |
| Stop execution normally | STOP | \* | \* | \* |
| Stop execution with error | ERROR | *literal | location* |  |  |

The *literal* value indicated in operand 1 of the ERROR instruction could be a numeric error code, a brief string literal, or a location containing a string literal.

* 1. **Other**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **OPERATOR** | **OPERAND 1** | **OPERAND 2** | **OPERAND 3** |
| No-op | NOOP | \* | \* | \* |

1. **Sample Program**

The following sample program, though not very meaningful, tries to test all aspects of the language, assuming correct syntax. It also gives you an idea of what a program looks like. Feel free to critique, improve, etc.

PROGRAM TestAll

## This comment is just here to test comments

DATA

## This is where you would place Data Definitions

CONST ONE INT 1 1

VAR ACC INT 0 1

VAR TEMP1 INT 1

VAR TEMP2 INT

VAR ARRAY1 INT 10

VAR ARRAY2 INT 20 100

VAR STR1 STRING

CONST GREET1 STRING 5 Hello

CONST GREET2 STRING 0 Bye

CONST ERROR STRING 0 \*

An error occurred!

## Continued on next page

CODE

## This is where Executable Instructions go

## Assigning to ONE should give an error because it’s CONST

## Assignments to variables and arrays tests

ASSIGN 99 TEMP1

ASSIGN ONE TEMP2

RARRAY ARRAY1 0 TEMP1

RARRAY ARRAY1 ONE TEMP2

WARRAY 86 ARRAY2 0

WARRAY TEMP1 ARRAY2 TEMP2

## Arithmetic tests

ADD ONE TEMP1 ACC

ADD 10 5 ACC

SUB 20 ONE TEMP1

SUB ACC 8 ACC

MUL ACC 3 TEMP2

MUL 444444444 ONE TEMP2

DIV TEMP2 2 TEMP2

DIV 1000 ONE TEMP2

## Branching tests

JEQL ONE 1 NEXT1

START JEQL 1 ONE NEXT2

NEXT1 JLESS 0 ONE NEXT3

NEXT2 JLESS ACC -111 NEXT4

NEXT3 JGRTR 0 ONE START

NEXT4 JUMP \* NEXT5

## Looping, input & output tests

WRITE “Loop1”

NEXT5 ASSIGN 0 TEMP1

LOOP1 ASSIGN 4 TEMP2

WRITE TEMP1

LOOP TEMP1 5 LOOP1

WRITE “Loop2”

ASSIGN 0 TEMP1

LOOP2 ASSIGN 4 TEMP2

READ \* \* ACC

WRITE ACC

LOOP TEMP1 TEMP2 LOOP2

ASSIGN 0 TEMP1

LOOP3 ASSIGN 5 TEMP2

READFILE \* \* ACC

WRITE ACC

LOOP TEMP1 TEMP2 LOOP1

## Stack tests

PUSH “first”

PUSH “second”

POP TEMP2

WRITE TEMP2

POP TEMP1

WRITE TEMP1

POP

## Last should give error

NOOP

NOOP \* \* \*

## And finally … (only one of the following will

## work on any one run – comment out the others

ERROR 1

ERROR ERROR

STOP

## Not quite done yet!

## This is where input values go, if they are present at all

## Input (reserved word FINPUT and input values are optional)

## Only literals are allowed after FINPUT

## No comments, no whitespace!

FINPUT  
Hi

+10

-10

0

999999999