GPA Reference Manual

mbp2103: Michael Pierorazio, Project Manager
vaz2001: Vincenzo Zarrillo, Systems Integrator
dg2267: Dmitry Gimzelberg, Systems Architect
jmg2105: Juan Gutierrez, Quality Assurance

1. Introduction

GPA is a light weight, general purpose language whose design incorporates the low-level understanding of assembly language while allowing the use of high level constructs such as functions and loops. The premise behind GPA comes from programmers who are used to working close to hardware but require some high level constructs, and as a result, GPA falls somewhere between assembly code and the C language in terms of abstraction. GPA is simple to learn for those who know another language, but due to its high level syntax, it is also easy for beginners to understand. The goal of GPA is to be simple, powerful, and fast.

2. Lexical Conventions

2.1 Comments

GPA accepts two different types of comments:

2.1.1 Single Line Comments

The characters `//` begin a single line comment. Everything to the right side of these characters on the same line is ignored by the compiler.

2.1.2 Multiple Line Comments

The characters `/*` begin a multiple line comment and the characters `*/` will end the comment.

2.2 Identifiers

Identifiers are made of at least one letter and then may be followed by any number or combination of letters, numbers or underscores.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise (note: if the language has a shorthand equivalent defined for a keyword it is noted in the second column):

Keyword	Shorthand (if it exists)
variable	v
procedure	p
go	
return	r
loop	l
break	

```
if
elsif
else
print
println
```

2.4 Variables

There are two different types of objects stored by using the `variable` identifier.

2.4.1 Integers

An integer is a sequence of digits.

2.4.2 Strings

A string is a sequence of characters started and terminated by the single quote character (‘). Strings refer to an area of storage initialized with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end. Within a string a single quote must be preceded another single-quote (‘). Certain non-graphic characters may be escaped according to the following table:

‘	‘’
\	\\
BS	\b
NL	\n
CR	\r
HT	\t

3. Syntax Notation

The syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal and characters in `Courier.` Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript “opt,” so that

{ *expression*$_{opt}$ }

would indicate an optional expression in braces.

4. lvalues & procedures

There is only one type of lvalue in GPA which is an identifier.

`procedures` take at most three arguments of type integer and return an integer type only.

They are defined outside of the `go` procedure like this:

```
procedure foo(a, b, c)
```

```
{
  …
}
```

And are used within a procedure like this:

```
{ x =opt } foo(a, b, c)
```

5. Conversions
No conversions can be done because operations can only be done on either a single constant expression or a set of homogenous ‘variable’ therefore there is no need to do any conversions.

6. Expressions
The precedence of expression operators (see chart)

6.1 Primary expressions
Primary expressions involving function calls group left to right.

6.1.1 *identifier*
An identifier is a primary expression once it has been declared.

6.1.2 variable
An integer and a string are both primary expression each of their own respective type.

6.1.3 (*expression*)
A parenthesized expression is a primary expression whose type and value are identical the same expression without parentheses.

6.1.4 *primary-expression* ($expression\text{-}list_{opt}$)
A function call is a primary expression followed by an optional listing of comma-separated expressions which serve as the arguments to the function.

6.2 Unary operators
Expressions with unary operators group right-to-left (these operators only apply to variables of integer type).

6.2.1 – *expression*
The result of the ‘– ‘ operator is the negative of the *expression* and is the same type.

6.2.2 ! *expression*
The result of the ‘!’ operator is the one’s complement of the *expression*.

6.2.3 *expression* ++
The result of the postfix '++' operator is the value of the *expression* incremented by 1.

6.2.4 *expression* --
The result of the postfix '-- ' operator is the value of the *expression* decremened by 1.

6.3 Multiplicative Expressions
The multiplicative operators * and / group left-to-right (these operators only apply to variables of integer type).

6.3.1 *expression* * *expression*
The '*' operator yields the product of the two expressions.

6.3.2 *expression* / *expression*
The '/' operator yields the (integer) quotient of the two expressions.

6.4 Additive Operators
The additive operators + and – group left-to-right (these operators only apply to variables of integer type).

6.4.1 *expression* + *expression*
The '+' operator yields the sum of the two expressions.

6.4.2 *expression* – *expression*
The '-' operator yields the difference of the two expressions.

6.5 Shift Operators
The shift operators << and >> group left-to-right (these operators only apply to variables of integer type).

7.5.1 *expression* << *expression*
7.5.2 *expression* >> *expression*

6.6 Relational Operators
The relational operators group left-to-right, but this is not useful because if "a<b" is true then "a<b<c" is breaks down to "1<c" which is probably not what a programmer wanted to write (these operators only apply to variables of integer type).

6.6.1 *expression* < *expression*
6.6.2 *expression* > *expression*
6.6.3 *expression* <= *expression*
6.6.4 *expression* >= *expression*

For each of these operators, the result is 0 if the relation between each *expression* is false, 1 if true.

6.7 Equality operators
These operators only apply to variables of integer type.

6.7.1 *expression* == *expression*
6.7.2 *expression* != *expression*
The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except that they have a lower precedence.

6.8 *expression* & *expression*
These operators only apply to variables of integer type.
The '&' operator groups from left-to-right and gives the bitwise 'and' of the two expressions.

6.9 *expression* | *expression*
These operators only apply to variables of integer type.
The '|' operator groups from left-to-right and gives the bitwise 'or' of the two expressions.

6.10 *expression* && *expression*
These operators only apply to variables of integer type.
The '&&' operator groups from left-to-right and yields 1 if both its operands are 1, otherwise, 0.

6.11 *expression* || *expression*
These operators only apply to variables of integer type.
The '||' operator groups from left-to-right and yields 1 if one or both of its operands are 1 and 0 otherwise.

6.12 *identifier* = *expression*
The value of the expression replaces the value of the identifier (if it has already been defined).

7. Declarations
Declarations have the form:

7.1 variable *identifier* = *expression, identifier* = *expression, …*
A set of variables can be created and defined in one line, separating each identifier with a comma.

7.2 variable *identifier* = *expression*
A single identifier is created and defined on the same line.

7.3 variable *identifier, identifier* …
A set of identifiers can be created all at once, separating each identifier name with a comma. These identifiers are assumed to be defined later.

7.4 variable *identifier*
A single identifier is created and is assumed to be defined later on.

8. Statements
Except as indicated, statements are executed in sequence.

8.1 expression statement
expression \n

8.2 compound statement
c*ompound-statement:*
{ statement-list }

s*tatement-list:*
statement
statement \n *statement-list*

8.3 Conditional statement
There are multiple forms of the conditional statement:

if (*expression*) *statement*
if (*expression*) *statement* elsif (*expression*) *statement* … else *statement*
if (*expression*) *statement* else *statement*
Both ‘if’ and ‘elsif’ must have their *expressions* evaluate to 1 in order for their respective *statements* (or *{ statement-list }*) to be executed, otherwise they jump down to the next ‘elsif,’ ‘else’, or continue. The ‘else’ ambiguity is resovled by connecting it to the last elseless ‘if.’

8.4 loop statement
The loop statement has the form:

`loop` *expression* (or null) : *expression* (or 1) : *expression* (or null)
statement

The loop statement can take up to three arguments (in this order): an initialization *expression*, a conditional *expression*, and finally a increment/decrement *expression.* The second and third *expression* are evaluated at the end of each loop. If no second *expression* is defined, the loop condition is assumed to always be true.

8.5 Break statement
The break statement causes termination of the smallest enclosing loop.
`break`

8.6 Continue statement

The continue statement causes control to pass to the end of the loop.

`continue`

8.7 return statement

The return statement can have two forms:

`return`

`return` *expression*

Both statements exit out of a function back to the calling procedure. The second form is used to return a value computed by the function that the calling procedure is using in an *expression*.

9. Scope Rules

Variables are local to the procedure in which they are declared unless defined in the go function, which makes them global.

10. Constant Expressions

All expressions evaluate to constants.

16. Examples

```
procedure foo(a, b)
{
  variable c=0
  if( a > b )
    return 0;
  loop : a <= b : a++
  {
    c=c+1
  }

  return c
}

procedure loopDemo(a, b)
{
  v z,d,i=0;
  d = 10;
  z = a + b - d;
  loop : z != b :
  {
    z--;
    if ( z <= 0 )
      break;
    else
      i++;
  }
}

procedure:gcd(a,b)
```

```
{
  if(a == 0 && b == 0)
    b = 1
  elsif (b == 0)
    b = a
  elsif (a != 0)
    loop : (a != b) :
    {
      if (a <b)
        b -= a
      else
        a -= b
    }
return b
}

go()
{
  variable a=4, b=9, c
  c=foo(a,b)
  if( c == 0)
    println 'b is less than or greater than a.'
  else
  {
    print 'The difference of a and b is '
    print c
    println '.'
  }

  loopDemo(c, a)

  v x=18, y=12, z
  z = gcd(x, y)
  print 'the gcd of '
  print x
  print 'and '
  print y
  print 'is '
  println z
}
```