COMS W4115 PROGRAMMING LANGUAGES AND TRANSLATORS

GBL Language Reference Manual

Yiqing Cui(yc3121) Sihao Zhang(sz2558) Ye Cao(yc3113) Shengtong Zhang(sz2539)

March 7, 2016

CONTENTS

1 Introduction

Most of us grows up with games, but it is extremely difficult to create a game from zero. We would like to provide such a language which could help game developers to generate their games easily and fast. So here comes Game Building Language (GBL). GBL is a user-friendly programming language for game building, and our compiler will compile it to LLVM. Game developers can use GBL to create any games based on coordinates easily, from Gomoku, Chess to Tactical War games (e.g. Fire Emblem), even Role-playing Game (Dragon Quest). The only thing that game developers need to do is providing the game rule, the map information, the picture of items and related settings. Besides, there are lots of build-in function and data type in GBL which could save lots of time for users. We are seeking to develop an intuitive and efficient language so users could focus more time on the design of games not on the technical process.
This document serves as a reference manual for the GBL.

With our proposed language, we aim to let users:

1. create the coordinate-based game easily.
2. intuitively learn and develop.
3. spend the least time doing most work.
4. have a wonderful time developing experience.

2 Syntax Notaions

Some special syntax notaions may be used to state the lexical and syntatic rules in this document:

1. Brackets [] enclose optional items.
2. Parenthesis () means the notation.
3. Asterisks * indicate that items could repeat zero or lots of times.
4. Dash - means an area from the the former to the latter.

3 Lexical Conventions

This chapter presents the lexical conventions of GBL, describing which tokens are valid, including comments, naming convention of identifiers, reserved keywords, operators, separators, new line and braces, and give the developers a right start to use GBL to avoid lexical errors.

3.1 Comments

Our language only supports single line comment. It is made with a leading # in the line:
Example:

```
# This is a comment
```

3.2 Identifiers

An identifier of GBL is a case-sensitive string which starts with a letter or an underscore. You could add an optionally characters which consist of letters, underscores and numbers. An identifier could not be any Keywords (see next subsection). The length of an identifier is from 1 to 256.
We could write a regular expression of an identifier as follows:

```
[a-z or A-Z or _] [a-z or A-Z or 0-9 or _](optional) (length of the
    identifier should be 1 to 256)
```

Here are some legal examples:

```
a a1 ab a_ abc1 _a _a1_
```

Here are some illegal examples:

```
1a %a a@ for int
```

3.3 Keywords

Our keywords are divided into six types:
1. Basic Data Keywords

```
int double bool string
```

2. Composite Data Keywords

```
list dict game player sprite map
```

3. Basic Flow Control Keywords

```
for if elif else break continue while end
```

4. Basic Logic Keywords

```
and or not
```

5. Basic Compare Keywords

```
geq leq gt lt is neq
```

6. Other keywords

```
fun print
```

3.4 Operators

An operator is a token which leads to an operation on one, two or three operands. We divide operators into six group. You can refer them in Chapter 5 Expressions and Operators.

1. Arithmetic Operators

```
+ - * / %
```

2. Assignment Operators

```
= += -= *= /= %=
```

3. Bitwise Operators

```
& | ^ ~ >> <<
```

4. Domain Operators

```
@
```

5. Parenthesis Operators

```
[] () {}
```

6. Comment Operators

```
#
```

3.5 Separators

We use separators to separate tokens. The separators includs space, () and [].

3.6 New Line

A complete statement ends with an explicit carriage return. Also, we can use \ in the end to support multi-line statement. Example:

```
a = \
1
```

which means a = 1.

3.7 Braces

We use braces to separator different sections, noting a section's start and end. When using some keywords like if, else and for, users must include the braces to state the area. Example:

```
if a == 1
{
        b = 1
}
else
{
        b = 0
}
```

4 Types

4.1 Primitive Types

4.1.1 int

The *int* date type is singed 32 bit type integral number which is expressed in decimal. And it ranges from -2^{31} to $2^{31}-1$.
Example:

```
int x = 12
int y = -40
```

4.1.2 double

The *double* date type is singed 64 bit type floating number which is expressed in decimal.
Example:

```
double x = 1.23
double y = -0.25
double z = 1.0
```

4.1.3 BOOL

The *bool* date type is 1 bit type logical value which is either true or false.
Example:

```
bool x = true
bool y = false
```

4.1.4 CHAR

The *bool* date type is 8 bit type ASCII character.
Example:

```
char x = "a"
char y = "b"
```

4.1.5 STRING

The *string* date type is sequence of 0 or more characters. And each character is stored by char type.
Example:

```
string x = ""
string y = "reference manual"
```

4.1.6 NULL

The *null* date type is symbol represents the instance is not exist.

4.1.7 ARRAY

The *array* date type is sequence of 0 or more elements. The data type of each element should be declared in advance.
example:

```
array<int> x = [1, 2, 3]
array<char> y = ["x", "y", "z"]
array<string> z = ["xy", "yz", "zx"]
```

4.2 STRUCTURE

Programmer can define their own structure with keyword *strt* and *identifier*. Structure can contain other data types, and the format of definition is as following:

strt identifier{type-declare-list}

identifier is named by programmer and will be used as new data type. *type-declare-list* is a sequence of type declarations for the members of the structure. So we can define new structure as following:

```
strt address
{
string country
string city
string street
int streetnumber
}
#define a new structure "address", which has attributes country, city,
    street and streetnumber
```

and then use the new structure as following:

```
address newaddress
#declare variable newaddress whose type is "address"
country@newaddress = "US"
city@newaddress = "New York"
street@newaddress = "Broadway"
streetnumber@newaddress = 1088
#assign value to attributes of newaddress
```

4.3 Game Types

4.3.1 Game

The game object, which has attributes such as player number, player list and sprite list. And it controls the status of the whole game. And it has following build-in functions:
1. setGameName(string) : set the name of game
Example:

```
setGameName("gobang")@Game
#set the game name as "gobang"
```

2. setPlayerNum(int) : set the player number of the game
Example:

```
setPlayerNum(2)@Game
#set the player number as 2
```

3. initializeGlobalValue(function) : use function as parameter. Programmer can use this function to initialize other variables in the game
Example:

```
intializeGlobalValue(initializeGlobal)@Game
#initializeGlobal is a function which was defined in advance
#call initializeGlobal function before game start
```

4. setOperate(function) : use function as parameter. Programmer can use this function to set what to do in the game
Example:

```
setOperate(gameOperate)@Game
#gameOperate is a function which was defined in advance
```

```
#call gameOperate function to execute each round of the game
```

5. addIslegal(function) : use function as parameter. Programmer can use this function to set how to judge if the action of sprite is legal
Example:

```
addIslegal(isLegal)@Game
#isLegal is a function which was defined in advance
#call isLegal function to judge if the action of sprite is legal
```

6. setIsFinished(function) : use function as parameter. Programmer can use this function to set how to judge if the game is finished
Example:

```
setIsFinished(finished)@Game
#finished is a function which was defined in advance
#call finished function to judge if the game is finished
```

7. setGameover(function) : use function as parameter. Programmer can use this function to set what to do after the game is complete
Example:

```
setGameover(gameover)@Game
#gameover is a function which was defined in advance
#call gameover function after the game is finished
```

8. operate() : execute the game-playing activity in the round
Example:

```
operate()@Game
#call the function which is set by setOperate function in each round
```

9. gameOver() : execute the game over activity
Example:

```
gameOver()@Game
#call the function which is set by setGameover function after game
    finished
```

10. setLastMove() : set the status of the game after one round
Example:

```
setLastMove(Sprite, int, int)@Game
#move the sprite to a new position
```

4.3.2 Player

The player object of the game, which has attributes such as name and texture, player can be controlled by the Game object and is used to record the action of each player. And it has following build-in functions:
1. initialize(string) : initialize the name of the player and return the Player instance
Example:

```
player1 = initialize("player1")@Player
#get a variable player1 whose data type is Player and name is "player1"
```

2. setOperate(function) : use function as parameter. Programmer can use this function to set what to do in player's round
Example:

```
setOperate(oneMove)@player1
#oneMove is a function which was defined in advance
#call oneMove function when game status comes to player1's turn
```

3. setBackground(string) : set the texture of the player
Example:

```
setBackground("player1.png")@player1
#set the background of player1 as "player1.png"
```

4. operate() : execute the game-playing activity of the player in the round
Example:

```
operate()@player1
#call the function which is set by setOperate function when game status
    comes to player1's turn
```

4.3.3 Sprite

Sprite is atomic individual in the game, just as a piece. It has attributes such as texture, position and owner. And it has following build-in functions:
1. initialize(string) : initialize the name of the sprite and return the Sprite instance
Example:

```
sprite1 = initialize("chess")@Sprite
#get a variable sprite1 whose data type is Sprite and name is "chess"
```

2. setBackground(string) : set the texture of the sprite
Example:

```
setBackground("chess.png")@sprite1
#set the background of sprite1 as "chess.png"
```

3. setPosition(int, int) : set the position of the sprite
Example:

```
setPosition(1, 1)@sprite1
#set the position of sprite1 as (1, 1)
```

4.3.4 Map

The map object of the game, which has attributes such as size and texture, also it can store sprites data. And it has following build-in functions:

1. setSize(int, int) : set the size of map
Example:

```
setSize(10, 10)@Map
#set the size of map as 10*10
```

2. setBackground(string) : set the texture of map
Example:

```
setBackground("map.png")@Map
#set the background of map as "map.png"
```

3. addSprite(Sprite) : add a sprite to the map
Example:

```
addSprite(sprite1)@Map
#add the instance sprite1 whose data type is Sprite to the map
```

4. removeSprite(Sprite) : remove a sprite from the map
Example:

```
removeSprite(sprite1)@Map
#remove the instance sprite1 whose data type is Sprite from the map
```

5 Expressions and Operators

This section describes the expression and operators in GBL.

5.1 Expressions

Expression is a combination of at least one operand and zero or more operators, in which operand can be constant, variables, objects and so on, operators can be one of +, -, *, /, % or other function call. Some expressions examples are listed as follow

```
120
34 * 56 + 9
12.5 - - c
func(37, a)
```

Also, expressions can be grouped by parentheses, such as

```
(5+7) / (4+6)
```

Note that in above expression, the meaning of parentheses is different of which in function call.

5.2 Arithmetic operators

GBL supports standard arithmetic operators, including addition, subtraction, multiplication, division, module and negation. Note that for a specific binary operator, the two operands should be the same size. Here are some examples:

```
5 - 3               # subtraction, result is 2
y * 5               # multiplication, result is 5y, in which y must be a
    integer
9 / 4               # division, returns the quotient part, which is 2
2 + - 1.2           # error, the first operand is an integer while the
    second one is float number
25 % 7              # module, result is 4
```

5.3 Array Access operators

If we want to randomly access an element among an array, we can use [] operators. Note that for an array, the index starts from 0. For example

```
A[9]                # returns the 10th element of array A, whose index is
    9.
```

5.4 Comparison operators

GBL also supports different comparison operators, including geq, leq, gt, lt, is, neq. These operators are derived from the corresponding English words, which is very easy to understand. In fact, comparison is a kind of special binary operators in that the return value is always Boolean.

```
a gt b              # returns true if a is greater than b
a lt b              # returns true if a is less than b
a geq b             # returns true if a is greater than or equal to b
a leq b             # returns true if a is less than or equal to b
a is b              # returns true if a is equal to b
a neq b # returns true if a is not equal to b
```

5.5 Logical operators

GBL supports logical operators including and, or, not. They test the value of multiple conditions and returns a Boolean value.

```
a > b and b > c             # returns true is a>b and b>c holds at the same
     time
a>b or b>c                  # returns true if a>b or b>c
not a<b             # returns true if a<b does not hold
```

5.6 Assignment operators

Like other programming languages, assignment operator s tores a particular value into a variable. Also, assignment operator has a return value, which is the assigned value itself. This enables the feature of 'consecutive assignment'. Here are some examples

```
A = 5           # stores 5 into variable A
d = foo(12)     # stores the return value of foo(12) into variable d
Z = y =  9      # stores 9 into y, stores y into Z
```

5.7 Member Access operators

In GBL, the member access operator is denoted by symbol '@'. The usage of this symbol is quite intuitive. Here are some examples

```
fstVal@Object = 12
```

5.8 Operator Precedence and Associative Property

When a complicated expression being evaluated, the different operator precedence and associative property will lead to significant different results. Therefore, the precedence and associative property must be well-defined to rule out all ambiguity. Following is the list of precedence, small number means higher precedence. Same precedence shares the same number. All evaluation within the same precedence are occurred from left to right unless

1. Function call
2. Member access operator
3. parentheses
4. Unary operator
5. Multiplication; division; module
6. Addition; subtraction
7. Comparison operators
8. Logical and; logical or
9. Logical not
10. Assignment operator

6 Statements

6.1 overview

In programming language, a statement is the smallest standalone element that express some action to be carried out. There are two kinds of statement, simple statement (usually in one line) or a compound statement (may contain statements as components).

6.2 ASSIGNMENT

Assignment statement is to assign a left value or right value to a left value, assign value to a right value is forbidden cause right value doesn't have a space in memory, the syntax is:

```
Lvalue = RValue
LValue = LValue
```

Example:

```
int i = 3
#Assign 3 to a integer value i
int j = i

#Assign variable i's value to j
3 = i
#Error, try to assign a right value
```

6.3 PRINT

print evaluates the expression and writes the resulting object to standard output as a string, and automatically create a new line, the syntax is:

```
print expression
```

Example:

```
print "Hello World"
#Output: Hello World
a=1
print a
#Output: 1
b=1
print a+b
#Output: 2
```

6.4 IF

If-Elif-Else statement is a conditional statement. It is expected to take one or multiple Boolean conditions, controlled by single If, multiple Elif and one Else. Then it evaluates each condition in sequence until it finds one satisfied condition or exits. The syntax is:

```
if condition
  {statements}
(elif condition
  {statements})*
(else
  {statements})?
```

Example:

```
int i=4
if i
```

```
{
  print i
  i--
}
#Output: 4321
```

6.5 WHILE

While statement is a conditional statement. It allows code to be executed repeatedly based on a given boolean condition. It evaluates the condition repeatedly until it finds one can't meet the condition. The syntax is:

```
while condition
  {statement}
```

Example:

```
int i = 3
#Assign 3 to a integer value i
int j = i
#Assign variable i's value to j
3 = i
#Error, try to assign a right value
```

6.6 FOR

The for statement provides a compact way to iterate over a range of values, it repeatedly loops until a particular condition is satisfied. The syntax is:

```
for element in list
  {statements}
```

Example:

```
int i = 3
#Assign 3 to a integer value i
int j = i
#Assign variable i's value to j
3 = i
#Error, try to assign a right value
```

6.7 CONTINUE

The continue statement causes a jump to the end of current loop, including for and while loop. Example:

```
#In while loop:
int i=9
while i>=0
{
  if i==5
  {
    continue
  }
  i--
  print i
}
#Output: 987643210

#In for loop:
for i in range(10)
{
  if i==5
  {
    continue
  }
  print i
}
#Output: 012346789
```

6.8 BREAK

The continue statement causes a jump to the end of current loop, including for and while loop. Example:

```
#In while loop:
int i=9
while i>=0
{
  if i==5
  {
    break
  }
  i--
  print i
}
#Output: 9876

#In for loop:
for i in range(10)
{
  if i==5
  {
    break
  }
  print i
}
#Output: 01234
```

6.9 RETURN

A return statement causes execution to leave the current subroutine and resume at the point in the code immediately after where the subroutine was called, known as its return address. A function is allowed to specify a return value to be passed back to the code that called the function.

```
return true
return 1
return [1,2,3,4,5]
```

Example:

```
int func1()
{
  return 1
}

void func2()
{
  return 2
}

int main()
{
  print func1()
  #Output: 1
  print func2()
  #Error: return type doesn't match the function's return type
  return 0
}
```

7 FUNCTIONS

7.1 FUNCTION DEFINITION

Function can be defined with its name and return type. The format of function definition is as following:

return-type function-name (type-declare-list) {function-body}

return-type should in assist with the return type of *function-body*. For function with no return value, programmer can use key word *func* to define it.
Here are examples of function definition:

```
func printwords(string v1, string v2)
{
 print v1
 print v2
}
#define function to print two strings

int add(int v1, int v2)
```

```
{
 return v1 + v2
}
#define function to add two integers
```

7.2 FUNCTION CALLS

Functions can be called with its name and appropriate parameters. The format of function calling is as following:

function-name(parameter-1, parameter-2, parameter-3, ...)

Also for function which has return value, we can use it as a sub expression and the format is as following:

variable = function-name(parameter-1, parameter-2, parameter-3, ...)

Here are examples of function calling:

```
printwords("start", "end")
#print string start and end
res = add(1, 2)
#add integer 1 and 2
```

7.3 RECURSION

Function can call itself to do some complex things. The example for function recursion is as following:

```
int fact(int val)
{
 if(val == 1)
 {
  return 1
 }
 return val * fact(val - 1)
}
```

8 PROGRAM STRUCTURE AND SCOPE

8.1 PROGRAM STRUCTURE

GBL's program must exist in a single source file, with a ".gbl" extension. It consists of a number of functions declarations and exactly one main function as the program entrance.
The position of function declarations in the source code doesn't matter, that means a function can be called first and defined later in the source code.
Example:

```
int func1()
{
  return 1
}

string func2()
{
  return ": Hello World"
}

int main(){
  print func1()
  print func2()
  return 0
}
#Output: 1: Hello World
```

8.2 SCOPE

Variables in GBL declared within a block is accessible only within that block and blocks enclosed by it, and only after the point of declaration. A block is defined by braces"", including function block, conditional block, for-loop block, while-loop block. A global variable is defined in main scope which is out of all braces.
Example:

```
int func1()
{
  int i=5
  int j=10
  return i+j
}
int i=10

int main(){
  print func1()
  #Output: 15
  print i
  #Output: 10
  print j
  #Error: variable 'j' not found
  return 0
}
```