GOBLIN

Language Reference Manual

Kevin Xiao, kkx2101

Bayard Neville, ban2117

Gabriel Uribe, gu2124

Christina Floristean, cf2469

Intro

Goblin is a language designed to allow anyone to make their own turn-based game without extensive knowledge of software development. It will follow a simplified object oriented model that hews as closely as possible to the familiar way things act in real life. All programs create a playable turn based game where a user is placed on a rectangular world created by ASCII characters. There may be multiple rectangular worlds accessed from the initial one via doors, portals, or hatches. There is a feed of recent actions and their results as well as a section for displaying stats like health or money. All games compile to LLVM and follow the same basic loop structure:

Player moves
Print the map, action feed, and stats
Non-player characters move
Print the map, action feed, and stats

Lexical Conventions

Identifiers

Identifiers are strings used for naming variables, functions, entities, behaviors, and worlds. They can contain letters, numbers, and underscores, but must start with a letter. Identifiers are case sensitive.

Keywords

Keywords are case sensitive, are reserved for the language and cannot be used as identifiers:

and	behaviors	bool	build	controls	does
else	entities	false	for	if	is
not	null	num	or	player	return
string	this	true	void	while	worlds

Literals

String Literals

String literals are a sequence of zero or more letters, numbers, or other ASCII characters. These literals must be enclosed in single or double quotes.

Number Literals

Number literals are an optional minus sign followed by a sequence of decimal digits.

Delimiters

Spaces, tabs, and returns separate tokens and help format the program. These are used to help the user with readability; however, they are completely ignored by the compiler.

Comments

Comments are enclosed by /* and */. They are purely for user documentation and are ignored by the compiler.

Parentheses and Braces

Parentheses are used to enclose the arguments of a function or to alter precedence of expression evaluation (See Expressions and Operators below). Braces are used to compartmentalize code blocks under definitions like entities, behaviors, worlds, etc.

Commas

Commas separate function arguments.

Semicolons

Semicolons punctuate the end of expressions, declarations, initializations, and assignments.

Types

Primitive Data Types	**Definition**
Number	A 32 bit signed integer with a range from -2,147,483,648 to 2,147,483,647. It is similar to the integer type in C. The keyword is num.
Boolean	A 1 bit true or false value. The keyword is bool.

String	A sequence of characters enclosed by either double or single quotes. It is represented as an array of characters. The keyword is string.

Non-Primitive Data Types	**Definition**
Entity	A datatype representing a player, character, or object. An entity must be initialized with an ASCII character to represent it in the world. Built-in fields: x y solid
Player	A subtype of entity. It is a specific entity that can be controlled by the game player and contains a “controls” block.
World	A datatype representing a map that entities are placed in. The map is represented by a 2 dimensional array of stacks, and it must be initialized with x and y dimensions. Built-in fields: x y

Inheritance

New data types cannot be created but existing ones can be extended with the “is” keyword:

```
V:vulture is bird
```

Expressions and Operators

Operators with equal precedence are evaluated from left to right. Otherwise expressions are evaluated from highest to lowest precedence. Parenthesis can be used to create subexpressions, which have their own scope of precedence. This allows the programmer to control the order an expression is evaluated. Expressions can be nested using parentheses and are evaluated from the innermost subexpression outwards. If there are no nested expressions, subexpressions are evaluated from left to right.

Operators

1. Arithmetic Operators (multiplication, division, modulo, addition, subtraction):

```
*, /, %, +, -
```

Goblin supports 5 arithmetic operators in the traditional order of precedence and associativity. The multiplication, division, addition and subtraction operators are associative; the modulo operator is non-associative.. Multiplication, division, and modulo have equal precedence and a greater precedence than addition and subtraction, which also have equal precedence.

2. Logical Operators:

```
and, or, not
```

Logical operators operate on Boolean values or expressions which evaluate to Boolean values. Expressions that use these operators can be used with parenthesis, as described above. The logical operators `and` and `or` are associative. However, `not` is a unary operator, meaning it only takes one expression. `not` operates on the expression to the right of the operator.

3. Assignment (Assign, Assign and increment, Assign and decrement):

```
=, +=, -=
```

`Assignment` operators are used to assign a value to a variable, such as:

```
num number = 9;
```

Variables are strongly typed, meaning the type of the variable must be declared when created. However, it is unnecessary and syntactically invalid to specify the type every time the variable is used. It is possible to set a variable to the evaluation of an expression, such as:

```
number = 4 + 3; /*also an example of re-assigning a variable*/
```

Increment and assign and decrement and assign are used in a similar way, either incrementing the number value of variable or decrementing the number value of a variable by the number that is given as a result when the expression to the right of the operator is evaluated.

```
<variable>+=<value to increment by>

<variable>-=<value to decrement by>

number += 3; /*number, previously set to 7, is now equal to
10*/
```

`I`ncrement and assign and decrement and assign can only be used with variables of type num because it is meaningless to perform an increment or decrement operation on non-numeric

types. Furthermore the term to the right of the operator must be of type num because incrementing a number with a non-number gives an undefined result and is therefore not allowed. Parenthesis can be used on the right side of the operator as long as the type of the evaluated expression corresponds to the type declared for the variable.

4. Boolean comparisons (Not equal to, Equal to, Less than, Greater than, Less than or equal to, Greater than or equal to):

```
!=, ==, <, >, <=, >=
```

These operators are used to compare two expressions; they return a third Boolean value derived from the inputs. The expressions must be comparable types that make sense to use these operators. For example, comparing if a string is greater than a num is undefined and therefore not allowed. However, comparing a boolean with a boolean or a num with a num is meaningful and therefore allowed.

Program Structure & Game Loop

A Goblin program consists of 3 blocks of code that are used to generate a single player game executable. At a very abstracted level a Goblin program looks something like this:

```
entities{
      <character>:player{...}
      ...
}

behaviors{...}

worlds{...}
```

We believe the best way to design a turn based game is in terms of these 3 categories. A game is a collection of *entities* that have *behaviors* and are placed in *worlds*.

Entities are defined within the entities block. Consider the following:

```
entities{
      <character>:player{...}
      <character>:<entity_name>{
                              build{...}
                              does{...}
                              …
                              }
}
```

Entities are bound - using the : operator - to a specific character that is used to represent instances of the entity on the game board. No two entities can be bound to the same character.

As seen above, it is possible for entities to contain a does block and a build block within their definition. The does block contains the behavior logic of the entity. The build block is a constructor for the object. Code contained within the build block will be executed exactly once at the entity's creation.

Behaviors within the does block are run every time it is that entity's turn. It is not required for an entity to do anything and there is no limit to how many behaviors can be listed within a does block. An entity is only allowed to define one does block.

Due to the fact that all Goblin programs compile to a game, it is a fair assumption that a player will always be present. Therefore a programmer is required to define an entity named in the entities block, although there is no requirement that the player be displayed on the screen. We make this requirement because persistent data(meaning data we want to store across world changes like health or money) is stored in the player object because a player is the only invariant between changing worlds and game states. Despite this subtlety, a player is still an entity albeit an important one. To emphasize this, Goblin requires that the player entity be the first entity defined in any entities block.

Below is a sample of the structure of a player entity:

```
<character>:player{
                does{...}
                controls{
                            <key>:<behavior>
                            …
                            }
}
```

Like all other entities, the player is represented by a unique specified character and can do things specified in its does block. However a noticeable difference between the player entity and all other entities is the controls block. Within the controls block keys are mapped to behaviors using the : operator with the syntax, similar to how symbols are assigned to entities:

```
<key>:<behavior>
```

Behaviors are triggered when the specified key is pressed. The same key cannot be bound to multiple behaviors. Player should be the only entity with a controls block because it is the player is the only entity controlled by the user, control blocks in any other non-user controlled entity would be meaningless.

When considering any interesting game, game objects (meaning any object used to create the game - worlds or entities) have *behaviors*. Behaviors are defined within the behaviors block and can be thought of as similar to functions in other programming languages. Behaviors are intended to be general and reusable, although in practice this may not always be possible. For more information about defining behaviors, see below.

A Goblin game may contain many worlds or no worlds at all, depending on code within the worlds block, but a game containing no worlds would make for a very boring game. The initial world of a game is the world defined first in the worlds block. Like entities, world objects can contain a does block, which define behaviors that are run every time it is the world's (the computer's) turn and a build block, which functions as a constructor. Code required to generate a world generally will be contained within a build block. A world is only allowed to define one does block and one build block.

The Goblin compiler generates a turn based game as defined by the source code. However, there are some mechanisms of game operation that are invariant. Every Goblin game consists of two players, the user and the computer. Each player gets a turn and all turns must happen sequentially. This means that one player cannot take their turn until the previous player has completed their turn. Practically, this means that the game will wait indefinitely for the user to take their turn before allowing the computer to alter the states of any entity or world in the game because the computer will continue playing unless an end state of the game is reached or the player decides to quit. Despite the appearance that the player is given the first turn, the computer is always given the first turn in any game to allow for initial world creation and game setup. After each turn, regardless of if it's a player turn or a computer turn, the screen is redrawn to reflect changes to the game state.

The game loop can be described as follows:

1. Turn taken
2. Screen redraw

The progression of a turn taken by the computer is as follows: the actions for computer controlled objects (meaning on-screen entities and current the world) – defined in does blocks - are performed, the states of the computer controlled objects are updated, the screen is redrawn and presented to the user.

The progression of a turn taken by a user is as follows: the user presses a key which is mapped to a specific function corresponding to a behavior. When the key is pressed the programmer-specified behavior function is called and the screen is then redrawn.

The screen is redrawn automatically after every turn. A game loop will continue run unless the programmer defines a world that the player is sent to for certain conditions, such as winning or losing, to break the loop and end the game.

Statements

Basic Statements

The simplest type of statement is an expression followed by a semicolon.

```
<expression>;
print(“hi”);
a + b;
```

Return Statements

Return statements begin with “return” followed by an optional value, followed by a semicolon. Return statements inside behaviors return the return type of the behavior and exit the behavior. Return statements in a build or does block must not return a value, but still exit from the build or does block.

```
return <value>; /* In a behavior with return type matching type of value */
return 5; /* Inside a behavior with num return type */
return; /* Inside does or build block, or behavior with void return type */
```

Exit Statement

An exit statement quits the game.

```
exit;
```

For Loops

A for loop consists of a header and a body. The header consists of “for” followed by a parentheses-enclosed, semicolon-separated list of three expressions. The first expression is run once when before the first iteration of the loop and is optional. The second expression is the condition for the loop and is mandatory. The loop keeps running the code in the body until the condition evaluates to false. The third expression is run after each iteration in the loop and is optional. The body is enclosed in curly braces and consists of any number of statements.

```
for(<optional expression>;<condition expression>;<optional
expression>) {
      <statements>
}
for(i = 0; i < 10; i += 1) { /* i is previously declared */
```

```
      player.health -= 1; /* this line is run ten times */
}
```

While Loops

A while loop consists of a header and a body. The header consists of “while” followed by a parentheses-enclosed expression. The expression is the condition for the loop and is mandatory. The loop keeps running the code in the body until the condition evaluates to false. The body is enclosed in curly braces and consists of any number of statements.

```
while(<condition expression>) {
      <statements>
}
while(player.health < 10) {
      player.health += 1;
}
```

If Statements

An if statement consists of a header and a body. The header consists of “if” followed by a parentheses-enclosed expression. The expression is the condition for the if statement. The body is enclosed in curly braces and consists of any number of statements. If the condition evaluates to true then the statements in the body are executed.

```
if(<condition expression>) {
      <statements> /* Executed if the condition evaluates to true */
}
if(player.health > 10) {
      player.health = 10;
}
```

If Else Statements

If-else statements are the same as if statements, but the body of the if is followed by “else” which is followed by another body enclosed in curly braces. If the condition evaluates to true, then the statements in the first body are executed, like in the if statement, but if the condition evaluates to false, then the statements in the second body are executed. An else is attached to the most recent else-less if statement.

```
if(<condition expression>) {
      <statements> /* Executed if the condition evaluates to true */
} else {
      <statements> /* Executed if the condition evaluates to false */
}
```

```
if(player.health > 10) {
      player.health = 10;
} else {
      player.health += 1;
}
```

Else If Statements

By combining if and if-else statements it is possible to create a conditional with more than two possible results. The conditions can cascade into several different possible outcomes.

```
if(player.health > 10) {
      player.health = 10;
} else if(player.health == 5) {
      player.health = 1000;
} else {
      player.health = 100 /* player.health <= 10 and player.health !=
5 */
}
```

Declarations

Variable Types and Return Types

The simple variable types are num, bool, and string. The complex variable types are entity and any type that inherits from entity. All entities automatically inherit from entity. The return types for behaviors are the same as the variable types, but with the addition of void. A void return type means the behavior returns nothing.

Simple Variable Declarations

Simple variables are declared with a simple variable type, followed by the variable name, followed by “=”, followed by a literal value of the variables type, followed by a semicolon. Simple variables must be initialized with a constant literal value of their type when they are declared.

```
<simple variable type> <variable name> = <value matching the variable
type>;
num things = 10;
```

Complex Variable Declarations

Complex variables are declared with “entity” or the name of a user defined subtype of entity followed by the variable name followed by a semicolon.

```
<entity or custom subtype of entity> <variable name>;
```

```
goblin things; /* goblin is defined in entities */
```

Behavior Declarations

Behaviors consist of a header followed by a body. The header consists of a return type, the behavior name, and a parentheses-enclosed, comma-separated list of any number of parameters. Each parameter is a variable type and a parameter name. The parameter variables are available inside the behavior body when the behavior is called.

The body is enclosed in curly braces and consists of any number of variable declarations followed by any number of statements. If the return type is not void then there must be a return statement of a type that matches the return type. If the return type is void then a return statement is not necessary, but empty return statements are permitted.

```
<return type> <behavior name>(<var type> <parameter name>,…) {
      <local variable declarations…>
      <statements…>
      <return statement if return type is not void>
}
num getDifference(num a, num b) {
      num c = a - b;
      return c;
}
```

Field Declarations

Field declarations are the same as simple variable declarations, but they are at the beginning of player, entity, or world definitions and are owned by the player, entity, or world that they are defined inside. Fields start out with the literal value from their declaration whenever the player, entity, or world they are part of is instantiated. Field declarations may not be complex variable declarations.

Build Block

A build block is essentially a behavior that belongs to the player, an entity, or a world; takes no parameters; and has a void return type. The build block runs every time an instance of the player, entity, or world it is part of is created. For the player this occurs when the game starts. For an entity this occurs whenever an instance of it is placed on the board. For a world this occurs whenever it is loaded. All fields of the player, entity, or world that the build block is part of are accessible by name within the build body.

The build block consists of the word “build” followed by the body. The body is enclosed in curly braces and consists of any number of variable declarations followed by any number of statements.

```
build {
      <variable declarations>
      <statements>
}
build {
      num bonus = 5;
      health += bonus;
}
```

Does Block

A does block is essentially a behavior that belongs to the player, an entity, or a world; takes no parameters; and has a void return type. The does block runs once each turn for each instance of the player, entity, or world it is part of. For the player and the current world this is once each turn. For entities this is once each turn for each instance of the entity. All fields of the player, entity, or world that the does block is part of are accessible by name within the does body.

The does block consists of the word “does” followed by the body. The body is enclosed in curly braces and consists of any number of variable declarations followed by any number of statements.

```
does {
      <variable declarations>
      <statements>
}
does {
      num x = 10;
      heal(this, x);
}
```

Entity Definition

A entity declaration consists of a header then a body. The header begins with a single ASCII character, followed by a colon, followed by the entity name, optionally followed by an inheritance section. The character before the colon is used to represent the entity on the board. The inheritance section consists of “is” followed by another entity name. If the inheritance section is present, then the new entity will inherit all the fields and can be considered an instance of the entity after the “is”. If this section is not present, then the new entity inherits from the master entity. Inherited fields may not be overridden with a new field declaration with the same name. To change the value of an inherited field use a statement in the build block. The body is enclosed in curly braces and consists of field declarations, an optional build block, and an optional does block.

```
<entity symbol>:<entity name> <inheritance>{
```

```
      <field declarations>
      <optional build block>
      <optional does block>
}
g:goblin is monster{
      num money = 10;
      build {
            health = 5; /* Health inherited from monster */
      }
      does {
            move_towards_player(this);
      }
}
```

Player Definition

The player definition is the same as an entity definition with the addition of the mandatory controls block immediately after the field declarations. The controls block consists of the word “controls” followed by a curly brace-enclosed list of key mappings. Key mappings consist of a single ASCII character followed by a colon followed by a function call, i.e. `<ASCII character>:<function call>;`. Each key mapping means that when the ASCII character is received as input on the player’s turn, the function after the colon will be executed.

```
<player symbol>:player {
      <field declarations>
      <controls block>
      <optional build block>
      <optional does block>
}
@:player {
      num health = 10;
      controls {
            w:move_up(this);
      }
      build {
            health += random(0, 100);
      }
      does {
            health -= 1;
      }
}
```

World Definition

A world declaration consists of a header and a body. The header consists of the world name followed by “[“ then a num then “,” then another num then “]”. Square bracket enclosed numbers determines the shape of the world. The body is enclosed in curly braces and consists of field declarations, a mandatory build block, and an optional does block.

```
<world name>[<height>,<width>] {
      <field declarations>
      <build block>
      <optional does block>
}
castle[10,10] {
      num goblins_killed = 0;
      build {
            place(goblin, [2,3]);
      }
      does {
            move_towards_player(this);
      }
}
```

Built-in Functions

Place Function

```
void place(entity_name, num x, num y)
```

Places an instance of entity_name in the current world at the location specified by the “x” and “y” values.

Peek Function

```
entity peek(num x, num y)
```

Returns the top entity on the board at the given “x” and “y” values or null if there is nothing in the specified position.

Remove Function

```
bool remove(num x, num y)
```

Attempts to remove the top entity on the board at the given “x” and “y” values. It returns true if it removes something and false if there is nothing to remove.

Load Function

```
void load(world_name)
```

Closes the current world and opens the world “worldName” given as an argument. “world_name” is not a normal type name. It is the name from one of the world definitions. The build method for the new world is run when it is loaded.

Print Function

```
void print(string text)
```

Prints the given string to the log under the map display.

Random Function

```
num random(num start, num end);
```

Returns a random number between start and end.

Scope

Global Scope

A Goblin program contains the global variable “player”, which can be used to access attributes, such as player.health or player.money. Worlds is also a global variable that can be used to access the different worlds and their attributes, such as worlds.house.num_goblins. Other than player and worlds, no variables have global scope. The different types or entities, worlds, or behaviors defined in their respective code blocks are all ‘public’ in that they are visible in every part of the program.

This behavior function called handle_death shows how global variable ‘player’ might be used:

```
void handle_death(Monster me) {
        if me.health <= 0 {
           player.money += me.money;
        }
 }
```

Function Scope

Variables defined within a function are only visible and recognized within the function itself. The same logic applies to for loops, while loops, and if/else if/else blocks.

Standard Library

Types

Type	Description
Monster	General adversary of the player. The ‘attack’ variable is a num describing how much of a player’s health can be lost upon interaction. Built-in fields: num attack num health
Scenery	Inanimate surroundings in a game. The ‘durability’ variable is a num that determines how much strength a player would need to move or destroy it. Built-in fields: num durability
Trap	Inanimate surroundings that have a detrimental effect on a player (i.e. lower health). A user can define a behavior function to specify what affect the num variable ‘attack’ has on the player (health, money, etc.). Subclass of Scenery. Built-in fields: num attack

Functions

Function	Description	Usage

num distance(entity a, entity b)	Find the distance between two entities on the world grid.	`if (distance(me, player) < 10) {`
		`  move_towards_player(1);`
		`}`

bool adjacent(entity a, entity b)	Determine if two entities are adjacent.	`if (adjacent(me, player)) {`
		`  attack(me, player);`
		`}`

void move_left(entity e)
void move_right(entity e)
void move_up(entity e)
void move_down(entity e)
	Family of functions used to move an entity on the board.	`controls {`
		`  l:move_left(this);`
		`  r:move_right(this);`
		`  u:move_up(this);`
		`  d:move_down(this);`
		`}`

Example Program

```
entities {
    @:player {
        num health = 10;
        num money = 0;
        num attack = 2;
        num sword_strength = 7;
        controls {
            l:move_left(this);
            r:move_right(this);
            u:move_up(this);
            d:move_down(this);
            a:player_attack();
        }
        does {
            health += 1;
        }
    }
```

```
    m:monster {
        num health = 1;
        num attack = 1;
    }

    g:goblin is monster {
        /* inherits attack = 1 from monster */
        num money = 1; /* adds new money field */
        build {
             health = 3; /* overrides health = 1 from monster */
        }
        does { /* Each goblin does these behaviors in sequence each
turn */
            handle_death(this);
            chase_player(this);
        }
    }

    ~:ghost is monster {
        bool solid = false; /* solid = true is default unless
specified to
                            be false */
        does {
            wander(this);
        }
    }

    s:scenery {
       num durability = 1;
    }

    t:trap is scenery {
        num attack = 1;
    }

    O:rock is scenery {
       num attack = 0;
        num stone = 3; /* Player could receive this when breaking
stone */
        build {
            durability = 10;
        }
```

```
    }

    T:tree is scenery{
    }

    .:grass is scenery{
        bool solid = false;
    }

    D:door is scenery {
        bool solid = false;
        does {
            if (peek(this.x, this.y) == player) {
                load("dungeon");
            }
        }
    }

    ^:spike is trap {
        build {
            attack = 5;
        }
        does {
            dangerous_floor(this);
        }
    }

    #:lava is trap {
        bool solid = false;
        build {
            attack = 13;
        }
        does {
            dangerous_floor(this);
        }
    }
}

behaviors {
    void player_attack() {
         attack(player, peek(this.x, this.y + 1));
    }
    void dangerous_floor(trap t) {
```

```
        if (peek(t.x, t.y) == player) {
            attack(t, player);
        }
    }
    void chase_player(monster me) {
        if distance(me, player) < 10 {
            move_towards_player(me, 1);
        }
        if adjacent(me, player) {
            attack(me, player);
        }
    }

    void move_towards_player(monster me, num steps) {
       if (player.x > me.x) {
           me.x += steps;
       }
       else if (player.x < me.x) {
           me.x -= steps;
       }
       else {
           if (player.y > me.y) {
               me.y += steps;
           }
           else if (player.y < me.y) {
               me.y -= steps;
           }
       }
    }
    void handle_death(monster me) {
        if me.health <= 0 {
           player.money += me.money;
        }
    }
    void attack(monster attacker, player defender) {
        defender.health -= attacker.attack;
    }

    void wander(ghost g) {
        g.x += random(0, worlds.house.x) % worlds.house.x;
        g.y += random(0, worlds.house.y) % worlds.house.y;
    }
}
```

```
worlds {
    castle[15, 15] {
        num goblins_defeated = 0; /*Variable to keep world-specific
state*/
        build {
            place(goblin, 1, 2); /* built-in function to place
entities */
            place(player, 4, 9);
            place(door, 15, 15);
        }
        does {
            place(goblin, random(0, 15), random(0, 15));
        }
    }
    dungeon[10, 10] {
        num goblins_defeated = 0;
        num ghosts_defeated = 0;
        build {
            place(goblin, 1, 2); /* built-in function to place
entities */
            place(goblin, 3, 5);
            place(ghost, 7, 9);
           place(spike, 7, 8);
           place(spike, 7, 7);
           place(spike, 7, 6);
           place(lava, 6, 6);
           place(lava, 6, 7);
            place(player, 4, 9);
        }
        does {
            place(ghost, random(0, 10), random(0,10));
            place(goblin, random(0, 5), random(0, 10));
        }
    }
}
```