Craft - Language Reference Manual

Daniel Tal (dt2479) [Manager]
Martin Fagerhus (mf2967) [Language Guru]
Abhijeet Mehrotra (am4586) [System Architect]
Roy Prigat (rp2719) [Tester]

1. Lexical elements

a. Identifiers

An identifier, or name, is a sequence of letters, digits, and underscores (_). The first character cannot be a digit. Uppercase and lowercase letters are distinct (case-sensitive). Name length is unlimited. The terms identifier and name are used interchangeably.

b. Reserved Keywords and Symbols

`element`	`int`	`color`	`key_up`
`world`	`float`	`!!`	`key_down`
`event`	`bool`		`key_id`
`start`	`pair`		`events`
`reset`	`if`	`new`	`pos`
`def`	`else`	`delete`	`this`
`return`	`while`	`speed`	
`condition`	`action`	`angle`	`bounce`
`health`	`lives`	`direction`	`import`

c. Constants (as per C LRM)

i. Integer Constants

1. A sequence of digits is assumed to be a base 10 decimal number.
2. Digits 0 to 9 can be used
3. Ex. 654

ii. Real Number Constants

1. These are used to represent fractional (floating point) numbers.
2. Represented by a sequence of digits which represent the integer, a decimal point, and a sequence of digits to represent the fractional part.
3. Ex. 5.7

iii. String Constants

1. A string constant is a sequence of zero or more characters, digits, and escape characters.
2. Ex. “I am a string”
3. Ex. “\”I am a string with quotation marks\””

d. Operators

+, -	add, subtract
*, /, %	multiplication, division, modulo
=	assignment
>, >=, <, <=	inequality operators
==, !=	equal to, not equal to
&&, ||, !	not, and, or
.	access

e. **Delimiters**

   i. **Parentheses:** Used to show precedence in operational and expression evaluation, to enclose parameters within function calls, and as inseparable parts of our pair types.
   ii. **Commas:** Used to separate arguments in function calls and to separate values in pair data types.
   iii. **Semicolon:** Used to end statements.
   iv. **Curly Brackets:** Used to mark the start and end scope of functions, loops, conditionals, and world definitions.

f. **Whitespace**
   i. Only used to separate specific words/tokens.

g. **Comments**
   i. Only one line comments allowed using "#" (hashtag symbol).

2. **Data types**

a. **Primitive Data Types**
   i. **Integer Types**
      1. Numbers of Integer type will be declared *int*
      2. Syntax: int <name> = <integer number>;
      3. Ex: int a = 123;
   ii. **Floating Point Types**
      1. Fractional numbers will be declared as *float*
      2. Syntax: float <name> = <fractional number>;
      3. Ex: float a = 5.7;
   iii. **Boolean Types**
      1. Boolean values will be declared as *bool*
      2. A boolean value can be either *true* or *false*
      3. Syntax: bool <name> = <boolean value>;
      4. Ex: bool alive = false;

b. **Non-primitive Data Types**
   i. **Pair Types**
      1. *pair* is defined by two integer values, separated by a comma, and enclosed by parentheses.
      2. Anything except natural numbers (nonnegative) will be rejected as well as any pair values that exceed the game grid size.

3. Syntax: pair <name> = (int,int);
4. Ex: pair object = (100,100);
5. **Operations on Pair Types**
    a. **Addition**
        i. Syntax: pair <name> = <pair type> + <pair type>;
        ii. Ex:

```
pair pair_1 = (10,10);
pair pair_2 = (20,20);
pair new_pair = pair_1 +
pair_2;
# new_pair == (30,30)
```

    b. **Subtraction**
        i. Syntax: pair <name> = <pair type> - <pair type>;
        ii. Ex:

```
pair pair_1 = (10,10);
pair pair_2 = (20,20);
pair new_pair = pair_2 -
pair_1;
# new_pair == (10,10)
```

    c. **Multiplication**
        i. Syntax: pair <name> = <pair type> * <pair type>;
        ii. Ex:

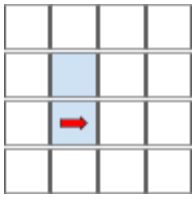
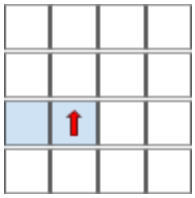
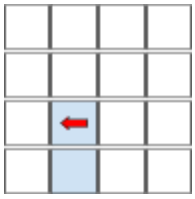
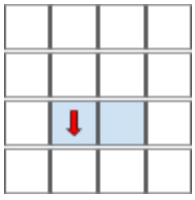
```
pair pair_1 = (10,10);
pair pair_2 = (20,20);
pair new_pair = pair_2 *
pair_1;
# new_pair == (200,200)
```

    d. **Division**
        i. Syntax: pair <name> = <pair type> / <pair type>;
        ii. Ex:

```
pair pair_1 = (10,10);
pair pair_2 = (20,20);
pair new_pair = pair_2 /
pair_1;
# new_pair == (2,2)
```

    e. For operations it is only allowed to calculate results which are natural numbers.

ii. Element Types

1. *element* is an object which is a part of the game's world.
    a. Rectangular shape
    b. Required attributes
    size, direction, speed, color, position(can also be passed as an argument at the time of object creation).
    c. Additional attributes are optional
    d. Size is described by a tuple, (x,y), supporting rectangular shapes
    e. Direction is the direction of the element
        i. Direction can be any number of degrees.
        ii. Initial support will be for 0, 90, 180, 270 degrees
        iii. Placement of the element on the grid will be bound to position of the element and it will rotate accordingly based on direction.
        iv. Examples below. The block, size==(1,2), is attached at position==(2,2) in a 4x4 world. The element is is placed at position (2,2) and situated on the grid based on direction.
        v.
        vi.  0 degrees
        vii.  90 degrees
        viii.  180 degrees
        ix.  270 degrees

2. Syntax:

```
element <name> {
      size = (x,y)
      direction = <int>;
      color = <hex>;
      speed = <int>;
}
```

3. Example:
    a.

```
element square_block {
      size = (2,2);
      direction = 0;
      color = ffffff;   # Black
      speed = 0;
}

#This will create a black square block
#size 2x2 (4 pixels)
#direction == 0, pointing at 0 degrees
#speed == 0, element not moving
```

3. Functions

a. Built-in functions:

Syntax	**Description**
`delete(element)`	Removes element from the world
`restart()`	Call the destructor (deletes/frees all memory and resets the world)
`add_event(event)`	The function adds the event passed into the parameter to the global event loop that runs in the global loop at every clock tick.

b. User-defined functions

i. Defining a function:

```
def function_name(args) {
      return
      return_element;
}
```

ii. Calling a function:

```
function_name(args);
```

4. Event blocks

a. Define events in the game with *event*

b. Syntax:

```
event (<element>) {
      condition {
            <some condition>
      } action {
            <some action that will happen if
condition == true>
}
```

c. Example:

```
event die(player p) {
      condition {
            p.health == 0;
      } action {
            p.lives = p.lives - 1;
            world.reset();
}
```

5. **Control Flow Statements**
    a. **Conditional statements**
        i. if/else statement:

```
if (<condition>)
{
    <statements>;
}
else {
    <statements>;
}
```

    b. **While loops**

```
while(<condition>) {
    <statements>;
}
```

6. **Program Structure and Scope**

In order to run a program, the program file must contain a main ‘world’ function. Standard files/libraries can be imported using ‘import’. The world function is the starting point of execution.

Each function/event/element within the file must be enclosed by curly brackets to determine its scope. It can be created/defined in the main file before the ‘world’ function and then called within ‘world’ in order to implement/use the function/event/element within the game world.

Furthermore any new instance of an element defined within the world function, is automatically added to the game world.

7. Sample Program

```
event die(player p) {
     condition {
          p.health == 0;
     } action {
          p.lives = p.lives - 1;
          world.reset();
}

event win(player p, treasure t) {
     condition {
          p !! t;  # collision
     } action {
          world.end();
     }
}

event moveUp(player p) {
          condition {
               key_down(upArrow);
          } action {
               p.direction = 90;
          }

event moveDown(player p) {
          condition {
               key_down(downArrow);
          } action {
               p.direction = 270;
          }

element wall {
     size = (2,1);
     direction = 0;
     color = ffffff;   # Black
     speed = 0;
}
```

```
element player {
      size = (1,1);
      direction = 0;
      color = f2333f;  # Blue
      health = 100;
      lives = 3;
      speed = 1;
}

element treasure {
      size = (1,1);
      speed = 0;
      direction = 90;
      color = 00ffff; # Yellow
}

world() {
      size = (100,100);
      player p1 = new player((0,0));
      treasure t = new treasure((9,9));
      wall w1= new wall(2,3);

      # add events to the game events loop, and bind them to specific
      # elements

      add_event(win(p,t));
      add_event(die(p));
      add_event(moveUp(p))
      add_event(moveDown(p));

      }
}
```