# Senet

## Language Reference Manual

**26th October 2015**

Lilia Nikolova
Maxim Sigalov
Dhruvkumar Motwani
Srihari Sridhar
Richard Muñoz

# 1. Overview

Past projects for Programming Languages and Translators have included languages for expressing the setup and flow of playing card games. Inspired by such languages, we propose to extend the domain to general, two-dimensional board games. Examples of games that might be expressed in our proposed language are tic-tac-toe, checkers, and chess. A similar idea has been investigated by Romein, Bal and Grune (1995), who described a language called *Multigame* that compiled to a parallel game playing program.[1] The authors focused their research on parallelized artificial intelligence to find optimal moving while playing games created in *Multigame*. In part due to this research focus, the authors restricted the group of games that could be described in *Multigame* to those with fixed-sized boards (thereby excluding card games) and to those where all players have perfect information.

We propose a similar language focused on simple expression of board games; however, the its compiler will create games that may be played interactively on the command line. The players will execute the game program of their choice after which they will be presented with prompts that navigate them through the game. We have named our new language *Senet* after one of the oldest-known board games, which traces its origins back to ancient Egypt. This document serves as a reference manual for the *Senet* language.

## 1.1 Goals

With our proposed language, we aim to provide:

1. Intuitive, relatively high-level expression of the setup and flow of board games;
2. Simple description of boards and pieces; and
3. Static, strong typing, and a mix of C and Python syntax to minimize the learning curve.

# 2. Lexical Conventions

There are 6 kinds of tokens: identifiers, keywords, integer literals, string literals, expression operators, and other separators. Blank, tab, and newline characters are only used to separate tokens, and are discarded by the scanner.

[1] J. Romein, H. Bal and D. Grune. (1995). Multigame - A Very High Level Language for Describing Board Games. ASCI 95, pp. 278-287.

## 2.1 Comments

Lines (or the remainder of a line) can be commented one at a time with #. Senet does not recognize multiline comments.

## 2.2 Identifiers

Identifiers consist of a sequence of letters, digits, and underscores.  They must begin with either a letter or underscore. *Senet* considers two identifiers differing only by case to be different.

## 2.3 Keywords

The following are reserved keywords in *Senet* and may not be used for any other meaning:

- `int`
- `str`
- `bool`
- `void`
- `list`
- `group`
- `and`
- `or`
- `not`
- `if`
- `else`
- `while`
- `for`
- `break`
- `continue`
- `return`
- `assert`
- `True`
- `False`
- `None`

## 2.4 Literals

*Senet* includes four kinds of literals that have fixed values: `integer`, `string`, and `list` literals. Notably, *Senet* does not support floating point literals.

### 2.4.1 Integer

An `integer-literal` is a sequence of digits. All integer literals in *Senet* are base 10.

### 2.4.2 String

A `string-literal` is a sequence of characters enclosed by double quotation marks. A double quotation mark inside a string must be written as "`\"`". A newline character inside a string must be written as "`\n`". A backslash character inside a string must be written as "`\\`".

### 2.4.3 List

A `list-literal` consists of elements of a common type separated by commas and enclosed in brackets. In addition, a `list-literal` may be the empty list of length zero (`[]`).

```
list-literal -> []
             |  [ list-literal ]
             |  [ list-type ]

list-type -> integer-literal-list
             |  str-literal-list

int-lit-list -> integer-literal
             |  int-lit-list , integer-literal

str-lit-list -> string-literal
             |  str-lit-list , str-literal
```

# 3. Meaning of Identifiers

Since *Senet* is a strongly-typed language, identifiers are associated with types. There are 5 kinds of identifiers in the language: basic types, derived types, group definitions, function definitions, and group instances.

## 3.1 Basic Types

*Senet* includes a number of basic types inspired from C and Python as shown in the table below.

| Basic Types | Meaning |
|---|---|
| `int` | 32-bit Integer |
| `str` | String |
| `bool` | Boolean (`True` or `False`) |
| `void` | type of `None`, a value used to represent the absence of a value |

## 3.2 Derived Types

The following table lists *Senet* types are derivatives (collections) of a basic type. The list type takes a basic type, `T`, which indicates the type of its elements.

| Derived Types | Meaning |
|---|---|
| `list<T>` | Linked lists; e.g. `list<int> a; a = [1, 2, 3];` |

## 3.3 Group Definitions

The language is object-oriented, with inheritance (but no multiple inheritance). New groups can be defined with the group keyword as described in Section 5. *Senet* includes the standard groups shown in the table below built-in and meant to be extended by the programmer.

| Standard groups | Meaning | Built-in Methods & Attributes |
|---|---|---|
| `Object` | Base object group; all groups extend this | `Object()`: default constructor, which contains no statements. |
| `Board` | Defines board geometries, win conditions, cleanup methods | `remove(int x)`: removes the piece at index `x`<br><br>`owns(int x)`: returns the number of the player at index `x` of `cells`<br><br>`full()`: returns True if all spots are occupied, else False<br><br>`cells`: list of board cells, each element is |

| | | |
|---|---|---|
| | | either None or the piece at that spot<br><br>`toi(list<int> l)`: takes human-readable coordinate list, maps it to an internal cell index<br><br>`tol(int x)`: takes an internal cell index, and maps to a human-readable coordinate list |
| `Piece` | Defines possible moves, keeps track of position, owning player, and other needed variables | `place(Board b, int x)`: places the piece on board `b` at index `x` of `b.cells`<br><br>`owner`: the owner of the piece<br><br>`fixed`: whether or not the piece can be overwritten |

## 3.4 Group Instances

Identifiers with types corresponding to group definitions can be created as:

```
class identifier identifier ( expression-list ) ;

expression-list -> expression
                 | expression-list , expression
```

If the group's constructor requires parameters,the number of expressions matching the number of parameters must be included inside the parentheses.

## 3.5 Functions

An identifier is bound to a function using syntax described in Section 5. The identifier can then be used to call the function using a `call-expression`, which is described further in Section 6.

## 3.6 Boards Library

Senet comes with a standard library of Board subclasses:

| **Board Declaration** | **Meaning** |
|---|---|
| `Boards.Rect(int x, int y)` | `x` by `y` size rectangular board |
| `Boards.Loop(int x)` | Loop-shaped board with `x` cells |

| | |
|---|---|
| `Boards.Line(int x)` | Linear board with `x` cells |
| `Boards.Hex(int x)` | Hexagonal lattice of radius `x` |

# 4. Expressions

The precedence of expression operators in *Senet* follows the ordering of the following subsections 4.1 to 4.13, with the highest precedence first. All operators are left associative unless otherwise specified in the subsections below.

An expression is can be any of the following:

```
expression -> integer-literal
            |  string-literal
            |  list-literal
            |  identifier
            |  field-expression
            |  element-expression
            |  ( expression )
            |  binop-expression
            |  call-expression
            |  no-expression
            |  assign-expression
```

There are a number of binary operation (`binop-expression`) expressions. Because they have different precedence, the following binary operation productions will be discussed in separate subsections, along with `field-expression, element-expression, call-expression`, and `assign-expression`.

```
binop-expression -> multiplicative-expression
                 |  additive-expression
                 |  relational-expression
                 |  equality-expression
                 |  log-and-expression
                 |  log-or-expression
```

## 4.1 Primary Expressions

The most basic, and highest precedence expression are literals (`integer`, `string`, and `list`), identifiers, and parenthesized expressions. In addition, `no-expression` represents an empty expression.

## 4.2 Field Access

Access to an object's fields (methods or attributes) is accomplished using a period (`'.'`) between the object's identifier and the field as shown below. The type of the `field-expression` is the type of the accessed field.

```
field-expression -> identifier
                  | field-expression . identifier
```

## 4.3 List Element Access

A single element of a list can be accessed using an element-expression. The left expression must have type `list`. The right expression (in brackets) must have type `int`. The return type is the type of the list's elements.

```
element-expression -> expression [ expression ]
```

## 4.4 Function Call

Functions are called as follows, with return type equal to the called function's return type. The identifier must be a function.

```
call-expression -> identifier ( expression-list )
```

## 4.5 Unary Minus

The language includes the unary minus (`'-'`) operator that negates the value to its right. The expression operand must have type `int`. The operator is described by the following production rule:

```
unary-minus-expression -> - expression
```

## 4.6 Multiplicative Expressions

*Senet* also includes operators for multiplication (`'*'`), integer division (`'/'`), and modulo (`'%'`). The language does not support floating point arithmetic. The expression operands must be of type `int`. The value has type `int`.

```
multiplicative-expression -> expression * expression
                           | expression / expression
```

```
                        | expression % expression
```

## 4.7 Additive Expressions

Math operations in *Senet* are purely integral. The language includes operators for addition (`'+'`) and subtraction (`'-'`). The types of the expression operands must be `int`. The value has type `int`.

```
additive-expression -> expression + expression
                     | expression - expression
```

## 4.8 Relational Operators

*Senet* includes the following comparison operators: `>` (greater than), `<` (less than), `>=` (greater than or equal to), `<=` (less than or equal to). The operands must of of type `int`. The value has type `bool`.

```
relational-op -> > | < | >=  <=
```

```
relational-expression -> expression relational-op expression
```

## 4.9 Equality Operators

*Senet* includes the following equality operators `==` (equal to), `!=` (not equal to). The types of the operands must be the same. The value has type `bool`.

```
equality-expression -> expression == expression
                     | expression != expression
```

## 4.10 Logical NOT

The logical not operator negates its operand, which must be of type `bool`. The return type is `bool`.

```
log-not-expression -> not expression
```

## 4.11 Logical AND

The logical and operator takes two operands of type `bool`, and has a value of type `bool`.

```
log-and-expression -> expression and expression
```

## 4.12 Logical OR

The logical or operator takes two operands of type `bool`, and returns a `bool`.

```
log-or-expression -> expression or expression
```

## 4.13 Assignment Expressions

The assignment operator in the language (`'='`) is right associative. The value of an assignment expression is the value of the rightmost of the expressions it is composed of.

```
assign-expression -> identifier = expression
                   |  field-expression = expression
```

# 5. Declarations

Any given declaration may be one of the following:

```
declaration -> /* nothing */
             | declaration basic-declaration
             | declaration group-definition
             | declaration func-definition
```

Basic-typed variables are declared as follows:[2]

```
basic-declaration -> type-id identifier ;

type-id -> int | bool | str | void
         | identifier
         | list<type_id>
```

Group definitions follow, where the identifier in parenthesis must be a group from which the new groups is extended:

```
group-definition ->
           group identifier ( identifier )
                 { basic-declaration-list function-list-opt };
```

[2] If `type-id` is an `identifier` or `list<identifier>` the identifier has to be a defined group.

```
basic-declaration-list -> /* nothing */
                        | basic-declaration-list basic-declaration


function-list-opt -> /* nothing */
                   | function-list

function-list -> func-definition
               | function-list func-definition
```

Functions can be declared as follows:

```
func-definition ->
           type-id identifier ( formals )
                  { basic-declaration-list statement-list }
         | assert identifier ( formals )
                  { basic-declaration-list statement-list }


formals ->  /* nothing */
         | formals-list

formals-list -> type-id identifier
              | formals-list , type-id identifier
```

There is an additional special class of `assert` functions. This special class of functions evaluate a list of conditionals sequentially and returns `True` if all of them evaluate to true. If any of the statements evaluates to false, it returns a `False` and breaks immediately.

# 6. Statements

A statement is singular:

```
statement -> expression-statement
           | selection-statement
           | iteration-statement
           | jump-statement
           | board-mod-statement
```

A statement sequence allows many statements to be executed in order (left to right):

```
statement-list -> statement
                | statement-list statement
```

## 6.1 Expression Statement

An expression statement executes a single expression. The expression statement has a value equal to the expression's value:

```
expression-statement -> expression ;
```

## 6.2 Selection Statements

Branching on `if`, `elif`, and `else` are described by the following:

```
selection-statement ->
  if ( expression ) { statement-list }
| if ( expression ) { statement-list } elif { statement-list }
| if ( expression ) { statement-list } else { statement-list }
```

## 6.3 Iteration Statements

An iteration statement may be:

```
iteration-statement -> while-statement
                     | for-statement
```

A while loop statement executes the statement in brackets repeatedly as long as the expression in parentheses evaluates to `True`.

```
while-statement -> while ( expression ) { statement-list }
```

A for loop statement executes the statement in brackets repeatedly similarly to the while loop statement. However, the for loop statement requires a basic-typed variable to be declared, and it executes (incrementing the declared variable each time) until the variable is greater than the upper end of the range.

```
for-statement ->
           for (expression in { expression-list }) { statement-list }
```

```
expression-list -> expression
                 | expression-list , expression
```

## 6.4 Jump Statements

A jump statement may be one of the following:

```
jump-statement -> break ;
                | continue;
                | return expression ;
```

A `break` statement must be inside of a `for` or `while` loop. The effect of this statement is to terminate the loop and execute the next statement after the loop. A `continue` statement must be inside of a `for` or `while` loop. The effect of this statement is to jump to the begin of the next loop iteration. A `return` statement must be inside of a function block.

## 6.6 Board State Modification

The place (>>) and remove(<<) operators affect board state. The >> operator takes a `Piece` and places it on the board at a coordinate (described by a `list<int>`). Therefore, it requires its left operand to be derived from `Piece` and its right operands to be a `Board` and a `list<int>`, which describe the coordinate(s) on the board upon which to place the piece. Similarly, the << operator takes a piece off the board and places it into another `Board`. The left operand must be of type `Board` and the right operand must be of type `Piece`.

These operators automatically check to see if the move is legal before performing. If it is not legal, the statement has value `False`. Otherwise, it has value `True`.

```
board-mod-expression ->
     field-expression >> field-expression [ list<int> ]
     field-expression << field-expression [ list<int> ]
```

# 7. Program Structure and Scope

## 7.1 Structure

A *Senet* program consists of the following simple structure:

```
program -> setup-block turns-block EOF
```

Thus, all games must have two program sections: the `setup-block`, which contains global functions, groups, and parameters used to set up the game; and the `turns-block`, which contains a `function-list` which describe turn "phases" functions each of which operate as a "`while True`" loop but can call other functions in the `@turns` section. One of the phase functions must be named `begin`, this will be called when the game begins.

```
setup-block -> @setup { declaration-list statement-list }
turns-block -> @turns { function-list }
declaration-list -> /* nothing */
                  | declaration-list declaration
```

## 7.2 Scope

Identifiers declared in the `setup-block` are visible to the remainder of the program. Identifiers declared within functions are visible only to each function. Identifiers declared as part of groups can be accessed using the field access operator whenever the group instance can be accessed. Within a function that is within a group definition, other identifiers that are part of the group can be called via a field-expression using the `this` keyword. Instances of group definitions can only be created if the group definition can be accessed.

## 7.3 Other Built-Ins

A number of library functions are built into the language:

- `restart()`
- `exit()`
- `read(int)`
- `stoi(str)`
- `print(str)`

To begin the game over, use `restart`. To quit the program, use exit. To read `x` number of integers from standard input, use `read(x)`. To convert a string `s` to an integer, use `stoi(s)` -- note that `s` must be a string of one character only. To pass a string s to standard output, use `print(s)`.

In addition, two variables are built into the language to control the flow of games:

- `N_PLAYERS`
- `ON_MOVE`

N_PLAYERS sets the number of players for the game; default 1. ON_MOVE begins at 1 and at the end of each turn function, it is incremented up to N_PLAYERS. If it is incremented beyond N_PLAYERS, it is reset to 1 and the process begins again. ON_MOVE is used to know which player is currently taking his or her turn.

# 8. Example Program

Tic-tac-toe is a two-player game that is played on a three row, three column board. The players take turns placing either an “X” or an “O” in each cell. A player wins if three of their pieces fall in a line (vertical, horizontal, or diagonal). The game ends in a draw if all cells are full and no player has won. Below, we describe how our language could be used to create an interactive tic-tac-toe game.

```
@setup
{
      group b (Boards.Rect(3,3)) {

            three_in_a_row(int player) {
                  for (l in [[0,0], [1,0], [2,0]],
                              [[0,1], [1,1], [2,1]]
                              [[0,2], [1,2], [2,2]]
                              [[0,0], [0,1], [0,1]]
                              [[1,0], [1,1], [1,1]]
                              [[2,0], [2,1], [2,1]]
                              [[0,0], [1,1], [2,2]]
                              [[0,2], [1,1], [2,0]]  {
                                # Tests each line
                        if (this.owns(b.toi(l[0])) == player and
                              this.owns(b.toi(l[1])) == player and
                              this.owns(b.toi(l[2])) == player) {
                                    return True;
                        }
                        # owns checks the owner of a piece at an index,
                        # and returns -1 if the space is empty
                  }
                  return False;
            }

            bool won (int player) { # checks if the player won
                  if three_in_a_row(player)
                        return True;
                  return False;
            }

            bool draw() {
                  if this.full() {
                        return True;
                  }
```

```
                return False;
        }

        assert draw() {
                this.full();
        }
};



group Mark (Piece) {  # inherits from Piece
        fixed = True;   # piece cannot be overwritten

        str repr () {
                if (this.owner == 0) { # this.owner is the id of the owner
                        return 'X';
                } else {
                        return 'Y';
                }
        }
};


N_PLAYERS = 2; # number of players, this default of 1


reset()
{
        str c;
        str c = "";
        while(c != "y" && c != "n") {
                print("Do you want to continue playing?\n");
                print("Type y to continue and n to exit\n");
                c = read(1);
        }
        if(c == "y") {
                restart();  # restarts the game
        } else {
                exit();     # exits from the game
        }
}
}


@turns
{
begin () {
        # this is basically just "while True" with only 1 phase
        # players input moves by typing coordinates, e.g. "11" or "02"
        print("Input coordinates of square to place")
        print("in i.e. \"22\" or \"10\".\n"); # prompts the players
        int a = stoi(read(1)); # reads one character and converts it to an int
        int c = stoi(read(1));
```

```
	if (Mark >> B [a,c]) {
		if (B.won(ON_MOVE)) { #ON_MOVE is the index of the player
			print("Player " + itos(ON_MOVE + 1) + " wins.\n");
			print("Congratulations!");
			reset();
		}
		if (B.draw()) {
			print("Game ends in a draw.\n");
			reset();
		}
		# if the move was legal, went through successfully,
		# and the game is not over, pass the turn to the next player
	}
}
```