Sudoku Game Design Language (SGDL)

Language Reference Manual

Team Members:

Sijue Tan	st2669@columbia.edu
Yigang Zhang	yz2345@columbia.edu
Yu Shao	ys2528@columbia.edu
Rongzheng Yan	ry2213@columbia.edu
William Chan	wc2372@columbia.edu

COMS 4115 Programming Languages and Translators

Professor Stephen A. Edwards

October 22, 2010 Columbia University

1. Introudction

Nowadays, Sudoku game is becoming more and more popular aroung the world. Sudoku, oringinated in Japan, is a logic-based, number placement puzzle. The oringinal Sudoku game is based on a grid, which is comprised of nine columns and nine rows. Each row or column on the grid has all of the digits from one to nine. Most of the time, the grid is partially given. Sudoku players have to complete the puzzle based their mathmatic and logic reasoning skills.

The SGDL is a language designed to create a Sudoku Game. The language is easy for programmer to manipulate the matrix and the element in it, to set rules for the game and to examine the rationality of the initialization. SGDL's syntax is modeled after C/C++ and will be built on top of those languages. A SGDL format file will be the input file and the compiler will produce a C source file as output file that can then be compiled by any C/C++ compiler.SDGL saved Sudoku fans from spending their time to learn comlicate programming development techniques, what they only need is the SDGL language which is easily understandable, then they don't have to study anything else to design a Sudoku game. By using SDGL, user could create great Sudouku games using the built-in methods.

SDGL also maintain the potential of extending the language further to enable programmers to develop word games such as word search puzzles and crossword puzzles.

2. Lexical conventions

2.1 Comments

The token "/* " will introduce a comment. The "*/" will end the comment.

2.2 Key Words

`int`	`grid`	`while`
`bool`	`if-else`	`not`
`string`	`for`	`and`

```
 or                         Scan     (<data
 true                       type>)
false                       Random ( )
Print    (<data             left
type>)                      right
```

Random() is reserved for making sudoku game designs more extensible.

2.3 Identifier

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ‘‘_’’ counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

2.4 Constant

There are several kinds of constants, as follows:

2.4.1 Integer constants

An integer constant is a sequence of digits. The language only takes decimal.

2.4.2 String constants

String constants are sequence of characters surrounded by quote marks “”. Characters are also part of the string constants.

3 Types

3.1 Primitive Data Types

3.1.1 String

The String type is comprise of a sequence of charactrers. Also, strings are constant. Once you set up values for this type you cannot change them.

3.1.2 Integer

A basic type that contains a 32 bit signed integer, with a range of -2147483648 to 2147483647

3.1.3 Boolean

There are two possible values for the boolean type: either true or false. Also, true is any nonzero integer value and false can be represented as zero.

3.1.4 Array

An array type is a matrix of a primitive type.

Array (<arrayname>, <#row>, <#col>); // <#rol> refers to the number of rows to be created in the array, and <#col> refers to the number of columns to be created in the array.

Example: `Array (x, 2, 3); x ~ [ 1,2,3,2,3,4]; // This creates the array [ 1 2 3`

```
            2   3   4 ].
```

3.2 Object Type

3.2 .1 Grid

The Grid type represents the basic element of sudoku game

***Properties*:**

Cell (<row#>, <column#>) // refers to the cell specified by the parameters. Returns the value of that cell.

Example: `int a ; a ~ object.cell( 6,6)`;

Row (<row#>) // the value surrounded by the "()" refers to the specific row. Returns a one-dimensional array containing all the values in that row.

Example: `array (r, 1, 4); r ~ object.row(6)`;

Column (<column#>) // the value surrounded by the "()" refers to the specific column. Returns a one-dimensional array containing all the values in that column.

Example: `array (c, 4, 1); c~ object.column(6)`;

Diagonal (<left | right>) // refers to the two diagonals in the grid. "Left" refers the diagonal starting at the upper left-hand corner, while the "right" refers to the diagonal starting at the upper right-hand corner. Returns a one-dimensional array that contains all the values in that diagonal.

Example: `array (d, 1, 4);  d~ object.diagonal(left);`

Block (<row#>, <column#>) // sub-square of the grid. The block is referenced by the parameters indicating the location of the upper left corner cell in the block. Returns an array with the same dimensions as the block and containing the values of each cell in the block.

Example: `array (e, 3, 3);  e ~ object.block ( 3, 4 );`

Band (<band#>) // refers to a particular row of blocks. The value in the parentheses refers to the band number, starting with band 0 at the top and band n at the bottom. Returns an array with row size equal to the blocks' row size and the column size is the sum of all the column sizes of the blocks. The elements in the array will contain each cell's value, that includes each cell value, from upper left corner to the the bottom right corner, of each block.

Example: `array (f, 3, 9);  f ~ object.band(2);`

Stack (<stack#>) // refers to a particular column of blocks. The value in the parentheses refers to the stack number, starting with stack 0 on the left and stack n on the right. Returns an array with column size equal to the blocks' column size and the row size is the sum of all the row sizes of the blocks. The elements in the array will contain each cell's value, from the upper left corner to the the bottom right corner, of each block.

Example: `array (g, 9, 3);  g ~ object.stack(2);`

Method:

SetAsGiven (<false | true>): whether to show the value(s) in the specified cell(s). By default, all the cells in the grid are set to not visible.

IsRepeatable (<false | true>): whether the values in the specified cells can be duplicates.

IsVisible(<row#>,<column#>): whether the cell is visible. With value of ture or false.

DefineBlock (<rowsize>, <columnsize>): allows the programmer to define a sub-square of the grid. Its row and column dimensions must be factors of the grid's row and column dimensions respectively.

SumOfRows (<integer>):	the total that all the values in each row must sum to.
SumOfColumns (<integer>):	the total that all the values in each column must sum to.
SumOfDiagonals (<integer>):	the total that all the values in each diagonal must sum to.
SumOfBlocks (<integer>):	the total that all the values in each block must sum to.
MaxValueOfRows (<integer>):	what the maximum value of each row is permitted to be.
MaxValueOfColumns (<integer>):	what the maximum value of each column is permitted to be.
MaxValueOfDiagonals (<integer>):	what the maximum value of each diagonal is permitted to be.
MaxValueOfBlocks (<integer>):	what the maximum value of each block is permitted to be.
MinValueOfRows (<integer>):	what the minimum value of each row is permitted to be.
MinValueOfColumns (<integer>):	what the minimum value of each row is permitted to be.
MinValueOfDiagonals (<integer>):	what the minimum value of each diagonal is permitted to be.
MinValueOfBlocks (<integer>):	what the minimum value of each block is permitted to be.

4 Expression

4.1 Operator

4.1.1 Arithmetic Operator

'+' (binary, addition)

Example: `expression 1 + expression 2` // add the two expressions

'-' (binary, subtraction)

Example: `expression 1 - expression 2` // the fore expression subtracts the latter

'*' (binary, multiplication)

Example: `expression 1 * expression 2` // the fore expression multiplies the latter

‘/’ (binary, division)

Example: `expression 1 / expression 2` // the fore expression divides the latter

‘++’ (unary, postfix, first be used)

Example: expression_int ++   // increments integer variable by one

‘--‘ (unary, postfix, first be used)

Example: expression_int --   // then decrements integer variable by one

4.1.2 Comparison Operator

‘>’ (binary, greater than)

Example: `expression 1 > expression 2`

‘<’ (binary, less than)

Example: `expression 1 < expression 2`

‘=’ (binary, equal)

Example: `expression 1 = expression 2`

‘!=’ (binary, not equal)

Example: `expression 1 != expression 2`

‘>=’ (binary, greater than or equal to)

Example: `expression 1 >= expression 2`

‘<=’ (binary, less than or equal to)

Example: `expression 1 <= expression 2`

The comparison operators are defined as “greater than”, “less than”, “equal”, “not equal” etc.

4.1.3 Logical Operator

‘**not**’ (unary, prefix)

Example: *not expression_1*

‘**and**’ (binary)

Example: e*xpression 1 and expression 2*

‘**or**’ (binary)

Example: *expression 1 or expression 2*

4.1.4 Assignment Operator

‘~’ is an assignment operator in SGDL.

Example: *`Int a; a ~ 3;`*

4.1.5 I/O operation

‘scan’ is an input operation which get the input from the stantard input stream

Example: *`string x; x~scan(string)`*;

‘print’ is an output operation which output the values to the output stream.

Example: *`string x; x ~ ‟ Hello SGDL″ ; print (x);`*

4.2 Special Symbols

Parentheses “()” are used after while or if to state conditions; used after methods to state the arguments; used after a cell to indicate its location

Commas “,” are used to separate different arguments.

Braces “{}” are used for a collection of statment lists.

Double quotation marks “ “” “are used to state a string.

Semicolon “;” is used to terminate a statment.

Dot “.” is used only with grid object to access properties or methods.

5 Statement

STATEMENT is either Expression (Assignment, Method, Properties) | Loop | Condition .

5.1 Declaration/Assignment:

We only have one object type Grid which must be declared at least once. The syntax is shown below:

```
Grid (identifier, Argument, Argument);
```

We also define the declaration of primitive data types as the following syntax:

```
int a; string b; boolean c ;
```

5.2 Condition:

if: *'i*f' is the keyword used for conditional statement. If the condition associated with the 'if' statement is true, then the syntax is shown below:

Syntax:

```
if (condition)
{
/* statements to be executed */
}
```

else: 'else' is the keyword used in conjunction with 'if'. When the condition of the 'if' statement turned out to be false then the statement contains 'else' is executed. ‘else’ is matched with the nearest ‘if’ that doesn’t have ‘else’ statement.

Syntax:

```
else {
/* statements to be executed */
}
```

5.3 Looping Construct

The looping construct in SGDL is the keyword ***'for'*** and ***‘while’***.

5.3.1 For statement

The semantics of SGDL ***'for'*** is same as the C language *'for'* loop. It is used to execute the same piece of code till some condition is met.

Syntax:

```
for (initialization_expression 1; condition_expression 2;
looping times_expression 3)
{
/* statements to be executed till the termination condition is
reached */
}
```

5.3.2 While statement

The while statement has the form as follows:

Syntax:

```
while ( condition_expression )
{
/* statements to be executed */
}
```

The sub-statement is executed repeatedly so long as the value of the expression remains true (nonzero). The test takes place before each execution of the statement

5.4 Method

Methods obey the syntax shown below.

Syntax:

```
 Object.method (expression)
```

5.5 Break

"**break**;" statement terminates the innermost loop; can only be used in for-loops and while-loops

6 Sample Program

To create a sudouku game, there are several steps to be completed.

First , we declared an ID as a grid object with 9 columns and 9 rows. At the same time, we assigned values to each cell on the grid object.

Second, we defined blocks, some basic properties of the sudouku grid and decided which cells is visible. The rest of cells will be left to game players to fill out.

Third , we provided three types of sudoku games. The players can make their choices and begin to play the game. The Scan function will start to take players' input values which should be digits between one to nine. The Scane function will not stop taking input values until players type "-1" .

Finally, the program will check the input values with the oringinal values we gave at the very beginning.

The sample program of SGDL is shown below:

```
main ( )
{
    Grid (ClassicSudoku, 9, 9) ~ { 5, 3, 4, 6, 7, 8, 9, 1, 2,
                             6, 7, 2, 1, 9, 5, 3, 4, 8,
                             1, 9, 8, 3, 4, 2, 5, 6, 7,
                             8, 5, 9, 7, 6, 1, 4, 2, 3,
                             4, 2, 6, 8, 5, 3, 7, 9, 1,
                             7, 1, 3, 9, 2, 4, 8, 5, 6,
                             9, 6, 1, 5, 3, 7, 2, 8, 4,
                             2, 8, 7, 4, 1, 9, 6, 3, 5,
                             3, 4, 5, 2, 8, 6, 1, 7, 9 };

    for (int i = 0; i <= 6; i ~ i + 3)
    {
        for (int j = 0; j <= 6; j ~ j + 3)
        {
            ClassicSudoku.SetBlock ( i, j, 3, 3);
        }
    }

    ClassicSudoku.IsNotRepeatable;
    ClassicSudoku.MaxValueOfRows ( 9 );
    ClassicSudoku.MinValueOfRows ( 1 );
    ClassicSudoku.MaxValueOfColumns ( 9 );
    ClassicSudoku.MinValueOfColumns ( 1 );
    ClassicSudoku.MaxValueOfBlocks ( 9 );
    ClassicSudoku.MinValueOfBlocks ( 1 );
    ClassicSudoku.MakeVisible ( 0, 0 );
    ClassicSudoku.MakeVisible ( 0, 1 );
    ClassicSudoku.MakeVisible ( 0, 4 );
    ClassicSudoku.MakeVisible ( 1, 0 );
    ClassicSudoku.MakeVisible ( 1, 3 );
    ClassicSudoku.MakeVisible ( 1, 4 );
    ClassicSudoku.MakeVisible ( 1, 5 );
    ClassicSudoku.MakeVisible ( 2, 1 );
    ClassicSudoku.MakeVisible ( 2, 2 );
    ClassicSudoku.MakeVisible ( 2, 7 );
    ClassicSudoku.MakeVisible ( 3, 0 );
    ClassicSudoku.MakeVisible ( 3, 4 );
    ClassicSudoku.MakeVisible ( 3, 8 );
    ClassicSudoku.MakeVisible ( 4, 0 );
    ClassicSudoku.MakeVisible ( 4, 3 );
    ClassicSudoku.MakeVisible ( 4, 5 );
    ClassicSudoku.MakeVisible ( 4, 8 );
    ClassicSudoku.MakeVisible ( 5, 0 );
    ClassicSudoku.MakeVisible ( 5, 4 );
    ClassicSudoku.MakeVisible ( 5, 8 );
    ClassicSudoku.MakeVisible ( 6, 1 );
    ClassicSudoku.MakeVisible ( 6, 6 );
    ClassicSudoku.MakeVisible ( 6, 7 );
    ClassicSudoku.MakeVisible ( 7, 3 );
    ClassicSudoku.MakeVisible ( 7, 4 );
    ClassicSudoku.MakeVisible ( 7, 5 );
    ClassicSudoku.MakeVisible ( 7, 8 );
    ClassicSudoku.MakeVisible ( 8, 4 );
    ClassicSudoku.MakeVisible ( 8, 7 );
    ClassicSudoku.MakeVisible ( 8, 8 );

Grid (HyperSudoku, 9, 9) ~ { 9, 4, 6, 8, 3, 2, 7, 1, 5,
                             1, 5, 2, 6, 9, 7, 8, 3, 4,
                    7, 3, 8, 4, 5, 1, 2, 9, 6,
                    8, 1, 9, 7, 2, 6, 5, 4, 3,
                    4, 7, 5, 3, 1, 9, 6, 8, 2,
                    2, 6, 3, 5, 4, 8, 1, 7, 9,
                    3, 2, 7, 9, 8, 5, 4, 6, 1,
                    5, 8, 4, 1, 6, 3, 9, 2, 7,
                    6, 9, 1, 2, 7, 4, 3, 5, 8 };

for (int i = 0; i <= 6; i ~ i + 3)
{
    for (int j = 0; j <= 6; j ~ j + 3)
    {
        HyperSudoku.SetBlock ( i, j, 3, 3);
    }
}

for (int i = 1; i <= 5; i ~ i + 4)
{
    for (int j = 1; j <= 5; j ~ j + 4)
    {
        HyperSudoku.SetBlock ( i, j, 3, 3);
    }
}

HyperSudoku.IsNotRepeatable;
HyperSudoku.MaxValueOfRows ( 9 );
HyperSudoku.MinValueOfRows ( 1 );
HyperSudoku.MaxValueOfColumns ( 9 );
HyperSudoku.MinValueOfColumns ( 1 );
HyperSudoku.MaxValueOfBlocks ( 9 );
HyperSudoku.MinValueOfBlocks ( 1 );

HyperSudoku.MakeVisible ( 0, 7 );
HyperSudoku.MakeVisible ( 1, 2 );
HyperSudoku.MakeVisible ( 1, 7 );
HyperSudoku.MakeVisible ( 1, 8 );
HyperSudoku.MakeVisible ( 2, 4 );
HyperSudoku.MakeVisible ( 2, 5 );
HyperSudoku.MakeVisible ( 3, 5 );
HyperSudoku.MakeVisible ( 3, 6 );
HyperSudoku.MakeVisible ( 4, 1 );
HyperSudoku.MakeVisible ( 4, 3 );
HyperSudoku.MakeVisible ( 4, 7 );
HyperSudoku.MakeVisible ( 5, 2 );
HyperSudoku.MakeVisible ( 6, 4 );
HyperSudoku.MakeVisible ( 7, 0 );
HyperSudoku.MakeVisible ( 7, 1 );
HyperSudoku.MakeVisible ( 7, 6 );
HyperSudoku.MakeVisible ( 8, 0 );
HyperSudoku.MakeVisible ( 8, 1 );

Grid (DiagonalSudoku, 9, 9) ~ { 4, 7, 8, 1, 9, 5, 3, 2, 6,
                                5, 2, 6, 4, 8, 3, 7, 1, 9,
                                9, 3, 1, 6, 7, 2, 4, 5, 8,
                                2, 5, 9, 8, 6, 7, 1, 3, 4,
                                3, 6, 4, 2, 5, 1, 9, 8, 7,
                                1, 8, 7, 3, 4, 9, 5, 6, 2,
                                7, 1, 2, 9, 3, 8, 6, 4, 5,
                                6, 9, 3, 5, 2, 4, 8, 7, 1,
                                8, 4, 5, 7, 1, 6, 2, 9, 3 };

for (int i = 0; i <= 6; i ~ i + 3)
{
    for (int j = 0; j <= 6; j ~ j + 3)
    {
        DiagonalSudoku.SetBlock ( i, j, 3, 3);
    }
}
```

```
DiagonalSudoku.IsNotRepeatable;
DiagonalSudoku.MaxValueOfRows ( 9 );
DiagonalSudoku.MinValueOfRows ( 1 );
DiagonalSudoku.MaxValueOfColumns ( 9 );
DiagonalSudoku.MinValueOfColumns ( 1 );
DiagonalSudoku.MaxValueOfBlocks ( 9 );
DiagonalSudoku.MinValueOfBlocks ( 1 );
DiagonalSudoku.MaxValueOfDiagonals ( 9 );
DiagonalSudoku.MinValueOfDiagonals ( 1 );

DiagonalSudoku.MakeVisible ( 0, 1 );
DiagonalSudoku.MakeVisible ( 0, 4 );
DiagonalSudoku.MakeVisible ( 1, 3 );
DiagonalSudoku.MakeVisible ( 1, 5 );
DiagonalSudoku.MakeVisible ( 1, 8 );
DiagonalSudoku.MakeVisible ( 2, 2 );
DiagonalSudoku.MakeVisible ( 2, 5 );
DiagonalSudoku.MakeVisible ( 2, 6 );
DiagonalSudoku.MakeVisible ( 3, 1 );
DiagonalSudoku.MakeVisible ( 3, 2 );
DiagonalSudoku.MakeVisible ( 3, 3 );
DiagonalSudoku.MakeVisible ( 3, 5 );
DiagonalSudoku.MakeVisible ( 3, 7 );
DiagonalSudoku.MakeVisible ( 4, 0 );
DiagonalSudoku.MakeVisible ( 4, 4 );
DiagonalSudoku.MakeVisible ( 4, 8 );
DiagonalSudoku.MakeVisible ( 5, 1 );
DiagonalSudoku.MakeVisible ( 5, 3 );
DiagonalSudoku.MakeVisible ( 5, 5 );
DiagonalSudoku.MakeVisible ( 5, 6 );
DiagonalSudoku.MakeVisible ( 5, 7 );
DiagonalSudoku.MakeVisible ( 6, 2 );
DiagonalSudoku.MakeVisible ( 6, 3 );
DiagonalSudoku.MakeVisible ( 6, 6 );
DiagonalSudoku.MakeVisible ( 7, 0 );
DiagonalSudoku.MakeVisible ( 7, 3 );
DiagonalSudoku.MakeVisible ( 7, 5 );
DiagonalSudoku.MakeVisible ( 8, 4 );
DiagonalSudoku.MakeVisible ( 8, 7 );


/* Declare a field for the player to work in, and initialize to zero. */
Array (PlayerField, 9, 9);

for (int i ~ 0; i < 9; i++)
{
    for (int j ~ 0; j < 9; i++)
    {
        PlayerField[i, j] ~ 0;
    }
}


int game;
int row = 0;
int col = 0;
int val = 0;
int done = 0;
int wrong;

Print ("Welcome to the Sudoku Game!");

Print ("Which game would you like to play?";
Print ("1 - Classic Sudoku");
Print ("2 - Hyper Sudoku");
Print ("3 - Diagonal Sudoku");
Print ("Or type any other number to quit");
```

```
game ~ Scan (  );

while ( game >= 1 **or** game <= 3 )
{
    if ( game = 1 )
    {
        Print ( ClassicSudoku );
        for (int i ~ 0; i < 9; i++)
        {
            for (int j ~ 0; j < 9; j++)
            {
                if ( ClassicSudoku.Cell.IsVisible ( i, j ) )
                {
                    PlayerField[i, j] ~ ClassicSudoku.Cell ( i, j );
                }
            }
        }
    }
    else if ( x = 2 )
    {
        Print ( HyperSudoku );
        for (int i ~ 0; i < 9; i++)
        {
            for (int j ~ 0; j < 9; j++)
            {
                if ( HyperSudoku.Cell.IsVisible ( i, j ) )
                {
                    PlayerField[i, j] ~ HyperSudoku.Cell ( i, j );
                }
            }
        }
    }
    else
    {
        Print ( DiagonalSudoku );
        for (int i ~ 0; i < 9; i++)
        {
            for (int j ~ 0; j < 9; j++)
            {
                if ( DiagonalSudoku.Cell.IsVisible ( i, j ) )
                {
                    PlayerField[i, j] ~ DiagonalSudoku.Cell ( i, j );
                }
            }
        }
    }
    Print ( PlayerField );

    while ( done != -1 )
    {
        unless ( row => 1 and row <= 9)
        {
            Print ("Please enter the row of the cell you would like to edit the value of: ";
            row ~ Scan (  );
        {

        unless ( col => 1 and col <= 9 )
        {
            Print ("Please enter the column of the cell you would like to edit the value of: ";
            col ~ Scan (  );
        }

        unless ( val => 1 and val <= 9 )
        {
            Print ("Please enter the value you would like to put in that cell: ";
            val ~ Scan (  );
        }
```

```
            PlayerField[row - 1][col - 1] = val;
            Print ( PlayerField );

            Print ("Type -1 if you are done, or any other key if you want to edit additional
numbers:");
            done ~ Scan (  );
        }

        wrong = 0;
        for (int i = 0; i < 9; i++)
        {
            for (int j = 0; j < 9; j++)
            {
                if ( game = 1 )
                {
                    if ( ClassicSudoku.Cell ( i, j ) != PlayerField[i, j] )
                    {
                        wrong = 1;
                    }
                }
                else if ( game = 2 )
                {
                    if ( HyperSudoku.Cell ( i, j ) != PlayerField[i, j] )
                    {
                        wrong = 1;
                    }
                }
                else
                {
                    if ( DiagonalSudoku.Cell ( i, j ) != PlayerField[i, j] )
                    {
                        wrong = 1;
                    }
                }
            }
        }

        if ( wrong = 1 )
        {
            Print ("Wrong!  Please try the game again!");
        }
        else
        {
            Print ("That's right!");
        }

        Print ("Which game would you like to play?";
        Print ("1 - Classic Sudoku");
        Print ("2 - Hyper Sudoku");
        Print ("3 - Diagonal Sudoku");
        Print ("Or type any other number to quit");
        game ~ Scan (  );
    }

    Print ("Thanks for playing the Sudoku Game!");
}
```