COMS 4115 Programming Languages and Translators

Sudoku Game Design Language (SGDL)

Final Report

Team Members:

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

Professor Stephen A. Edwards

Dec 20, 2010 Columbia University

Table of Contents

Chapter 1
Introduction

The Sudoku game is a logic-based, number-placement puzzle that is a special form of Latin Squares. It is most often a puzzle played on a 9 x 9 grid that can only contain numbers. The player is normally required to fill each cell of the board with numbers one to nine, following specified rules. The rules are normally that each number cannot appear more than once on a row, on a column, and on a block. A block is normally defined as a 3 x 3 subset of the board, and there are nine blocks in the entire grid. In addition, the numbers along each row, column, and block must sum to 45. The starting gird will have certain numbers already filled in. It is believed that, at a minimum, 17 numbers must be exposed initially in order to produce a unique solution.

Although this form is the most popular version of Sudoku, there are many other variations, ranging from a standard Sudoku grid but with additional rules to unusual board arrangements like jigsaws and geometric shapes other than squares.

Although there have been various puzzles throughout history that were Sudoku-like, the creator of the modern version has been credited to Howard Garns, an architect from Indianapolis. He created them for Dell Magazine in 1979 and called them “Number Place.” Eventually, these puzzles made their way to a Japanese magazine in 1984. The magazine renamed these puzzles as Sudoku, which meant “single numbers”. It didn’t take long for Sudoku to become popular in Japan. But it was only when a British newspaper called “The Times” decided to publish the puzzle in 2004 that Sudoku became a worldwide phenomenon. Today, Sudoku puzzles are published in dozens of magazines in over 35 countries. There are also numerous books and computer games dedicated to this puzzle.

Our language, the Sudoku Game Design Language, is meant to help a game designer easily create Sudoku-related games.

Chapter 2
Tutorial

The sample program of SGDL is shown below.

```
generateNum(int row, int col)
begin
            int ranNum;
            int temp;
            print(num);
            ranNum~ Random(10-col);
            temp ~ num[10-col];
            num[10-col] ~ num[ranNum];
            print(num[10-col]);
            num[ranNum] ~ temp;
            return num[10-col] ;
end

Array num;                                              /*Global variables*/

main()                                                  /*Entry point of program */
begin

        int i; int j;int k;
        int a;
         int b;
        int col;
        int ran;
        string s;
        Array tempt1;
        num ~ [|1,2,3,4,5,6,7,8,9|];
        for (i~1;i<10;i++)
        begin
             Grid.Cell(1,i) ~ generateNum(1,i);
        end
        printg();
        print(Grid.Cell(1,1));
        for(j~2;j<10;j++)
        begin
           for(k~1;k<10;k++)
           begin
                if(k=9)
                begin
```

```
                    Grid.Cell(j,9) ~ Grid.Cell(j-1,1);
                end
                else
                begin
                    Grid.Cell(j,k) ~ Grid.Cell(j-1,k+1);
                end
            end
        end
print("----------------------------------------");
printg();
print("----------------------------------------");
tempt1 ~ Grid.Row(2);
Grid.Row(2) ~ Grid.Row(4);
Grid.Row(4) ~tempt1;

tempt1 ~ Grid.Row(3);
Grid.Row(3) ~ Grid.Row(7);
Grid.Row(7) ~tempt1;

tempt1 ~ Grid.Row(6);
Grid.Row(6) ~ Grid.Row(8);
Grid.Row(8) ~tempt1;

i~0;
while (i<60)
begin
a~Random(9);
b~Random(9);
print(a);
print(b);
Grid.MakeVisible(a,b);
i++;
end
printg();
print("----------------------------------------");
Grid.Cell(1,1) ~ 10;
Grid.SumOfRows(45);
s ~ "Hello world" ;
print(s) ;
print(Grid.Row(6));
Grid.CheckGrid ();
```

Chapter 3
Reference Manual

3.1 Lexical Conventions

SGDL is a programming language intended to create Sudoku games, with the capability to be extended to create any game related to grid as well. Syntax and coding style of SGDL are similar to C/C++. SGDL uses several built-in types and grid object. Especially, the grid object has properties and method just like classes in Java. Those properties provide the user a straightforward way to set up rules of Sudoku or grid games. Every SGDL program should have a main function. Everything declared outside of main function can be treated as global variable. Any function declarations are also located out of the main function's scope.

3.1.1 Key Words

main	for	false
int	while	Print (<data type>)
bool	not	Scan (<data type>)
string	and	Parse (<string>)
grid	or	Random (<seed>
if-elseif-else	true	

3.1.2 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.

3.1.3 Comments

The token “//” and “/* “ will introduce a comment. The “/*” will be terminated with the token “*/”. Especially, token “//” is just used for inline comments.

3.2 Data Types

3.2.1 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.

3.2.2 Integer

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

3.2.3 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.

3.2.4 Grid

The Grid type represents the basic elments of sudoku game

3.3 Expression

3.3.1 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 *expression 1* ++ // increments integer variable by one

'--' (unary, postfix, first be used)

Example: expression_int *expression 1* -- // then decrements integer variable by one

3.3.2 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.

3.3.3 Logical Operator

‘not’   (unary, prefix)

‘and’   (binary)

‘or’   (binary)

3.3.4 Special Symbols

Brakets “[ ]” are used after while or if to state condition; 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 lists.
White spaces and carriage return characters will be used to separate tokens.
Double quotation marks “ “” “are used to state a string.
Single quotation marks “ ‘’ “are used to state a character.

3.4 Statement

STATEMENT is either Declaration | Expression | Loop | Condition | Function | Method.

3.4.1 Declaration/Assignment:

We only have one object which must be declared. The syntax is shown below:

Type (Static) identifier_programmer (Argument, Argument);

3.4.2 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:

if (condition)
begin
/* statements to be executed */
end

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.

Syntax: else
begin
/* statements to be executed */
end

3.4.3 Looping Construct

The looping construct in SGDL is the keyword '*for' and 'while'*. 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; condition; looping times)
begin
Statement /* statements to be executed till the termination condition is reached */
end

3.4.4 While statement

The while statement has the form

while (*expression*)
begin
Statement /* statements to be executed till the termination condition is reached */
end

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

3.4.5 Method & Properties

Methods & properties obey the syntax shown below.
Identifier.method (expression)

3.4 Method & Properties

IsVisible (<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.
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.
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.

Other:
'.' (binary, used only with grid object to access properties or methods)
'~' (binary, assignment)

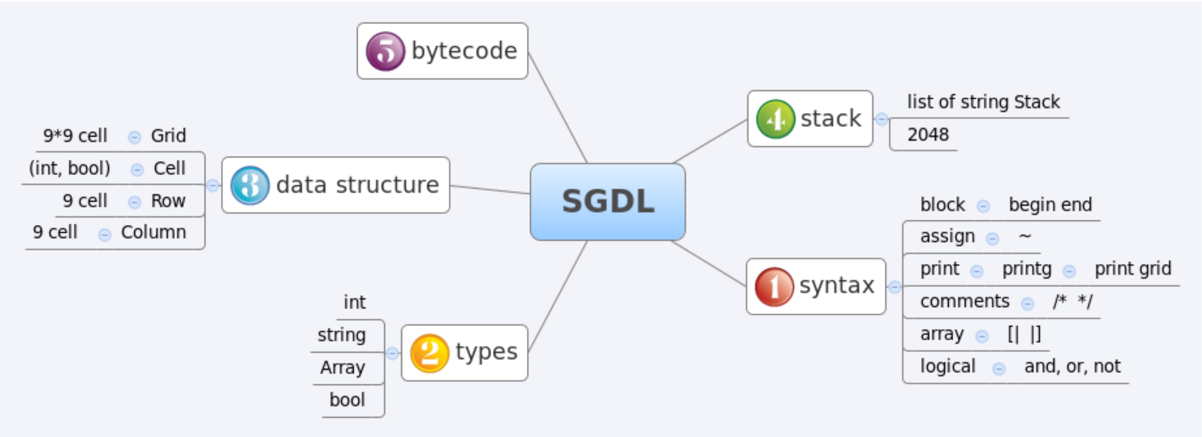
3.6 SCOPE & NAMES

3.6.1 Static Scoping

SGDL uses static scoping. That is, the scope of a variable is a function of the program text and is unrelated to the runtime call stack. In SGDL, the scope of a variable is the most immediately enclosing block, excluding any enclosed blocks where the variable has been re-declared.

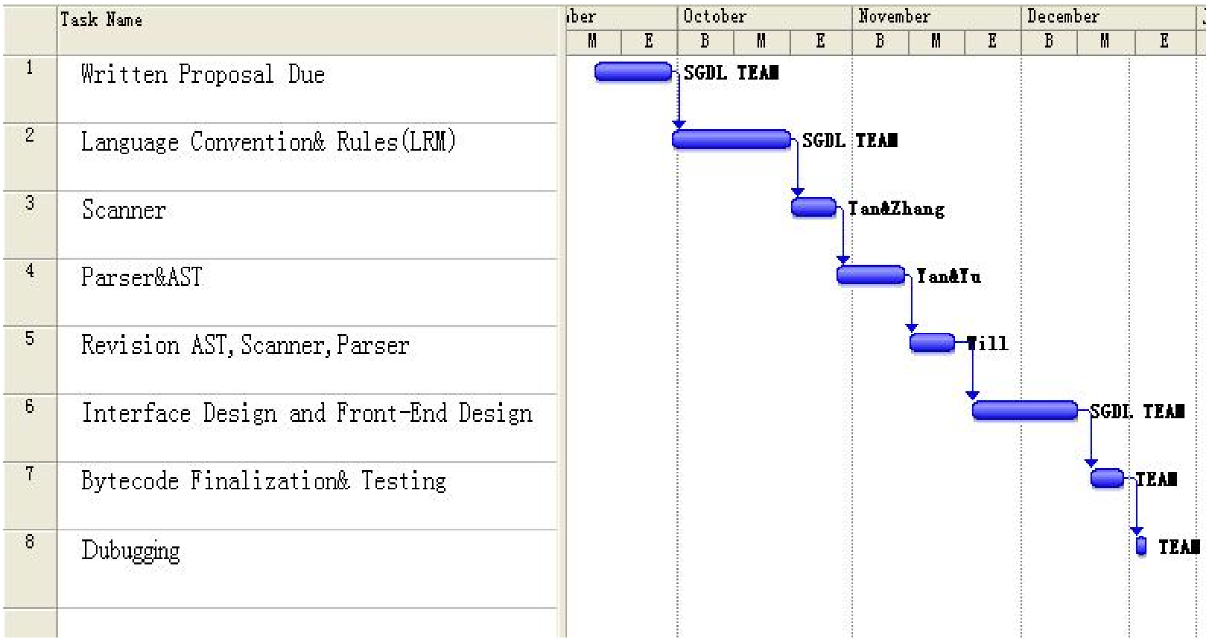
3.6.2 Global vs. Local

Global variable: The variables declared outside of the functions are global variables, which will be applied in the whole program except the function where there is a local variable with the same name as that of the global variable. Global variable will exist until the program terminates.
Local variable: The variables declared inside of the function are local variables, which will exist and be applied only inside that function.


bytecode
stack
list of string Stack
2048
9*9 cell
Grid
(int, bool)
Cell
9 cell
Row
9 cell
Column
data structure
SGDL
block
begin end
assign
~
print
printg
print grid
syntax
comments
/* */
array
[| |]
logical
and, or, not
int
string
Array
bool
types

Chapter 4
Project Plan

Project Timeline



Programming Style Guide

The syntax of SGDL is very similar to C/C++ language, which gives every team members an easier way to understand and design. At the very first of project, we used Google Docs to share ideas and to update useful files. We began to use Tortoise SVN since it is very fast to update several files at the same time. Every time a team member committed files to SVN repository, the one will inform the whole group by sending emails. We also hold team meeting every week so that we can work as a group.

Development Environments

O'caml: The entire project is implemented in Objective Caml Programming Language.
We used both Ubuntu Version and Windows Version of O'caml packets to design Sudoku Game Design Language.

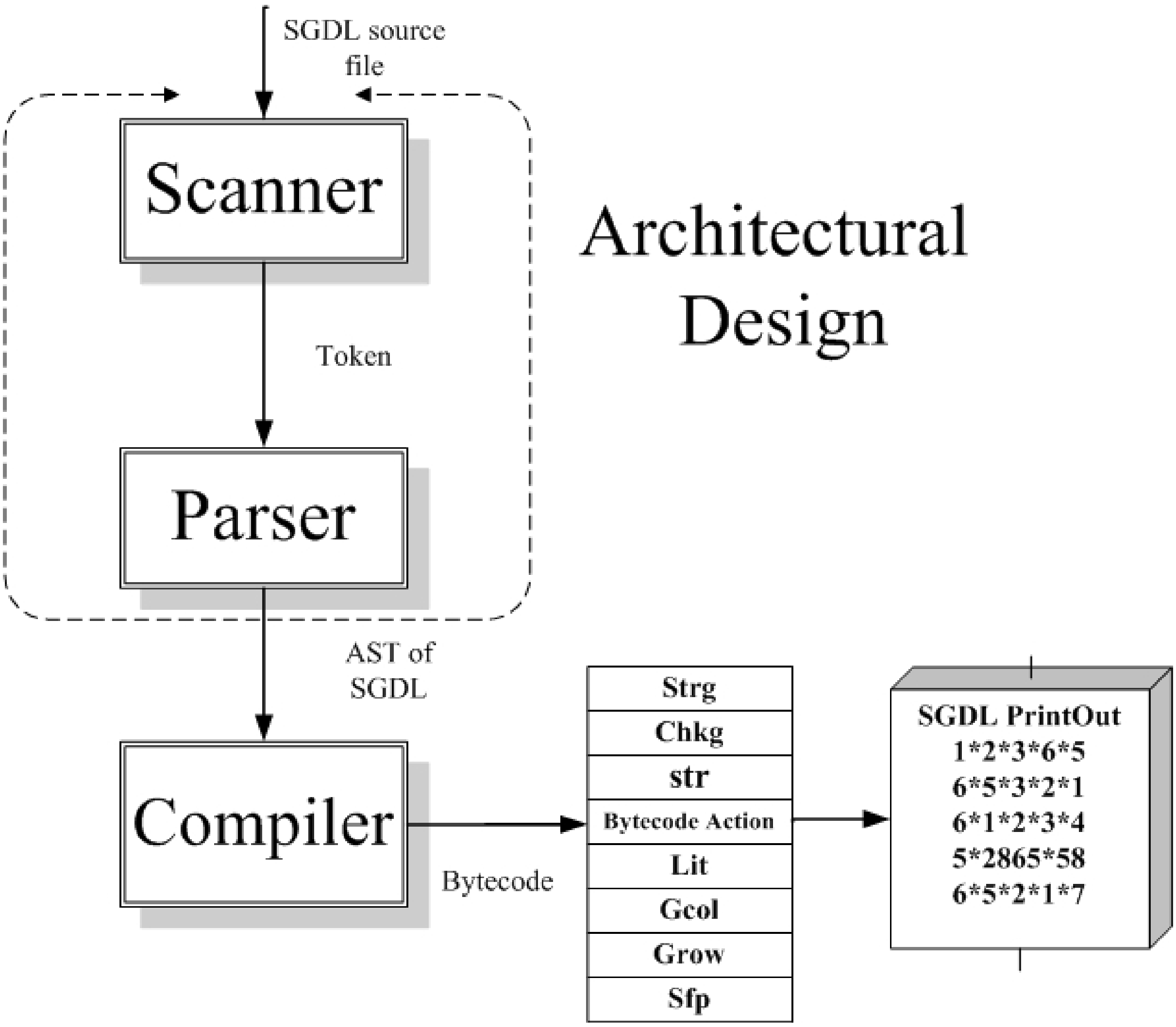
SVN: TortoiseSVN is version control / source control software for Windows. We used SVN to synchronize the code written by each team members. Also, the SVN works fine along with google code.

Project Log

Date:	***Objective:***
15 Sept 2010	Project Team Formation
22 Sept 2010	First Regular Team Meeting/SGDL Project Began First meeting with TA
29 Sept 2010	Written Proposal Due
5 Oct 2010	Discussing and Finalizing the Architecture of SGDL
11 Oct 2010	Finalizing Grammar/Job Distribution Scanner and Parser Design
27 Oct 2010	Language Reference Manual Due
27 Oct 2010	Finish rules of AST and Parser
1 Nov 2010	Interface Design and front-end Design
20 Nov 2010	Scanner and Parser vision II and Bytecode Studying
26 Nov 2010	Scanner and Parser Completed and Bytecode Design
10 Dec 2010	Testing Suite I completed and build shell script
18 Dec 2010	Bytecode and Execute file version I finished. Testing Suite II
19 Dec 2010	Debugging and working on Final Report
22 Dec 2010	Final Demonstration
22 Dec 2010	Project Report Due

Chapter 5
Architectual Design

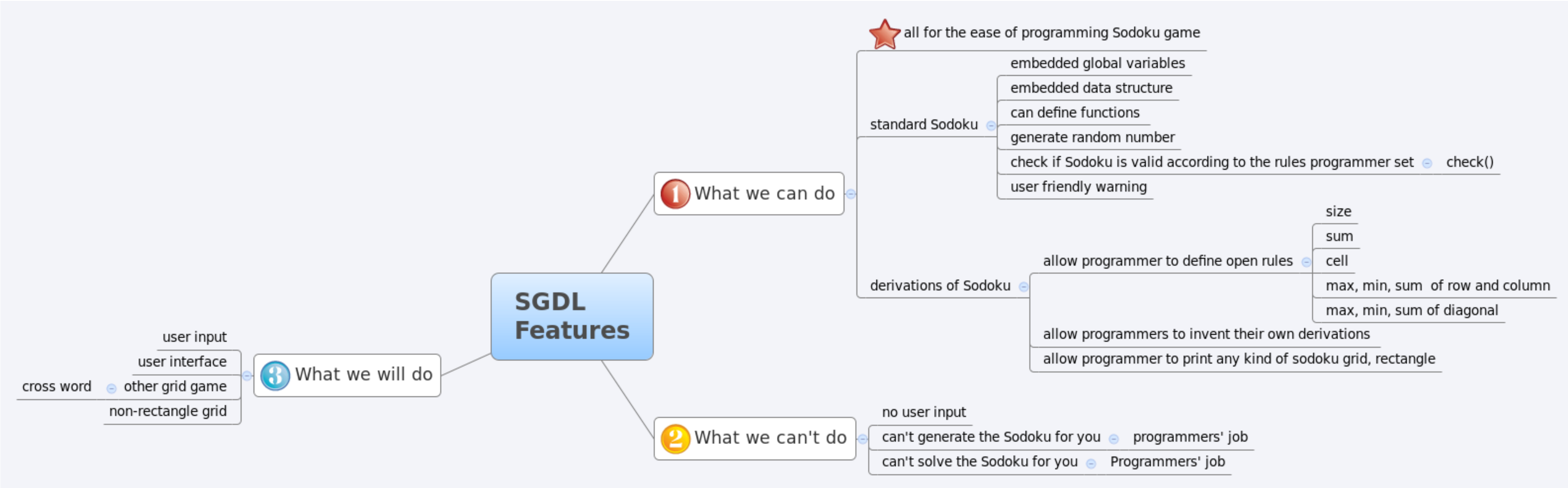
5.1 Components of SGDL Design



5.2 Description of SGDL

As shown in the block diagram above, SGDL has several major components such as scanner, parser , compiler and byte-code interpreter .
The scanner will take the SGDL source file created by programmer as input and perform lexical analysis of the input file.
The lexical analyzer will separate the input character stream and produces a stream of tokens.

The parser will process the tokens and analyses the structure of the program. It checks whether it conforms to the grammar of SGDL and then creates an AST.
Then the abstract syntax tree will be passed to the compiler. The compiler of SGDL will begins to find the entry point of the source SGDL program created by programmer and creates the symbol tables, which are checked in order to resolve variables and the types. For any invalid arguments, it will throw appropriate exceptions.


SGDL Features
What we can do
all for the ease of programming Sodoku game
standard Sodoku
embedded global variables
embedded data structure
can define functions
generate random number
check if Sodoku is valid according to the rules programmer set
check()
user friendly warning
derivations of Sodoku
allow programmer to define open rules
size
sum
cell
max, min, sum of row and column
max, min, sum of diagonal
allow programmers to invent their own derivations
allow programmer to print any kind of sodoku grid, rectangle
What we can't do
no user input
can't generate the Sodoku for you
programmers' job
can't solve the Sodoku for you
Programmers' job
What we will do
user input
user interface
other grid game
cross word
non-rectangle grid

5.3 Precedence and Associativity

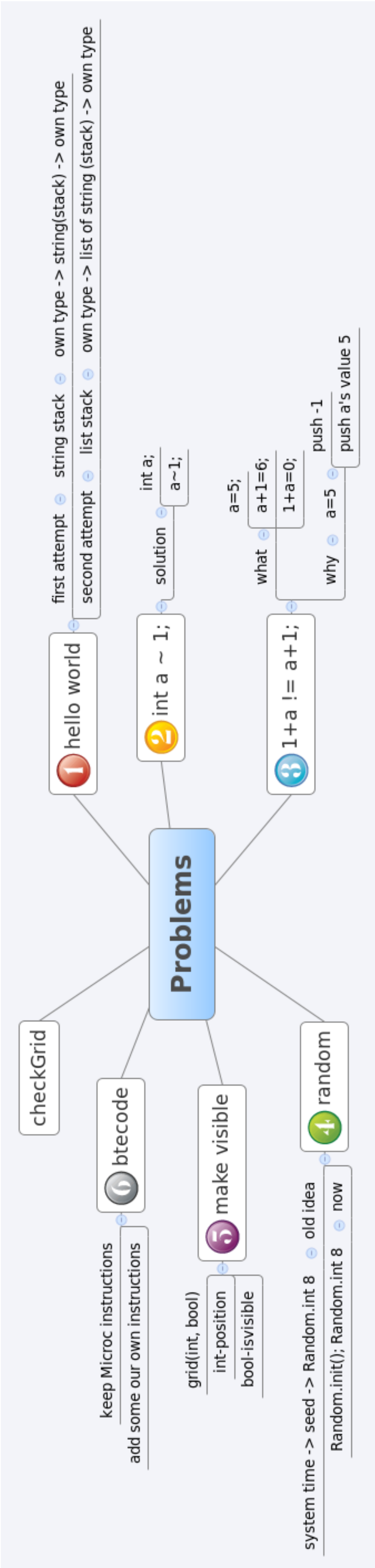
The precedence and associativity of the SGDL operators match those of C:

<table>
<tr><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>( )
[ ]
{ }
.
++
--</td><td>parentheses (function call)
brackets (array subscript)
braces (blocks)
dot operation
postfix increment
postfix decrement</td><td>left-to-right</td></tr>
<tr><td>!!</td><td>logical negation</td><td>right-to-left</td></tr>
<tr><td>*
/
%</td><td>multiplication
division
modulus</td><td>left-to-right</td></tr>
<tr><td>+
-</td><td>addition
subtraction</td><td>left-to-right</td></tr>
<tr><td><
>
<=
>=</td><td>relational less than
relational greater than
relational less than/equal
relational greater than/equal</td><td>left-to-right</td></tr>
<tr><td>==
!=</td><td>relational equal
relational not equal</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>logical and</td><td>left-to-right</td></tr>
<tr><td>||</td><td>logical or</td><td>left-to-right</td></tr>
<tr><td>~</td><td>assignment</td><td>right-to-left</td></tr>
<tr><td>,</td><td>comma (separate expressions)</td><td>left-to-right</td></tr>
</table>

Chapter 6
Test Plan

File	Role
Test-arith1.sl	Basic arithmetic between constants
Test_arith2.sl	Precedence, associativity
Test_arith3.sl	Basis arithmetic between variables
Test_array.sl	Type of array
Test_array1.sl	elements in an array
Test_array2.sl	Exchange between arrays
Test_array3.sl	Exchange elements between arrays
Test_cell.sl	Assignment of cells
Test_columntoarrays.sl	Assign a column to an array
Test_exchangeofcolumn.sl	Exchange between columns
Test_exchangeofrows.sl	Exchange between rows
Test_fib.sl	Recursion
Test_for1.sl	For basic loop
Test_for2.sl	For loop with grid
Test_func1.sl	User_defined function
Test_func2.sl	Argument_eval.order
Test_func3.sl	Argument_eval.order
Test_hello.sl	Hello world
Test_if1.sl	If statement
Test_if2.sl	else
Test_if3.sl	False predicate
Test_if4.sl	False else
Test_makevisible.sl	Makevisible()- arguments is constants
Test_makevisible2.sl	Makevisible()-arguments is variables
Test_maxofcolumns1.sl	MaxValueOfColumns():true
Test_maxofcolumn2.sl	MaxValueOfColumns():false
Test_maxofdiagnal1.sl	MaxValueOfDiagnals():true
Test_maxofdiagral2.sl	MaxValueOfDiagnals():left_false
Test_maxofdiagnal3.sl	MaxValueOfDiagnals():right_false
Test_minofrow1.sl	MinValueOfRows():true
Test_minofrow2.sl	MinValueOfRows():false
Test_minofcolumns1.sl	MinValueOfColumns():true
Test_minofcolumn2.sl	MinValueOfColumns():false
Test_minofdiagnal1.sl	MinValueOfDiagnals():true
Test_minofdiagral2.sl	MinValueOfDiagnals():left_false
Test_minofdiagnal3.sl	MinValueOfDiagnals():right_false
Test_minofrow1.sl	MinValueOfRows():true

Test_minofrow2.sl	MinValueOfRows():false
Test_ops1.sl	All binary operators
Test_ops2.sl	All logic operators
Test_printg.sl	Printg(): print an integrated grid
Test_random.sl	Generate the constants: 0-8
Test_Random.sl	Generate constants with certain limit
Test_rowtoarray.sl	Assign a row to array
Test_sumofcolumn.sl	SumOfColumn()
Test_sumofdiagnals.sl	SumOfDiagnals()
Test_sumofrows.sl	SumOfRows()
Test_var1_local2.sl	Local variables
Test_var1_local2.sl	Local variables
Test_var1_global.sl	Global variables
Test_while1.sl	Basic while loop
Test_while2.sl	While loop for grid
Test.sl	Generate an integrated Sudoku game
Test2.sl	Generate a special Sudoku grid
Test4.sl	Merry Christmas


Problems
hello world
first attempt
string stack
own type -> string(stack) -> own type
second attempt
list stack
own type -> list of string (stack) -> own type
int a ~ 1;
solution
int a;
a~1;
1+a != a+1;
what
a=5;
a+1=6;
1+a=0;
why
a=5
push -1
push a's value 5
random
system time -> seed -> Random.int 8
old idea
Random.init(); Random.int 8
now
make visible
grid(int, bool)
int-position
bool-isvisible
btecode
keep Microc instructions
add some our own instructions
checkGrid

Chapter 7
Lesson Learned

Sijue Tan

It is an exciting journey to design our own language and make it more and more complete. First of all, this course helps me learn how a special language goes through a compiler and finally be something the computer can understand. These phrases look so strange at beginning. However, it's very amazing after understanding how perfect they work with each. Besides, Ocaml is actually a hard language to understand because I am too familiar with Java's and C's language structure to accept one of such different language.

Secondly, using the version control system SVN is a wise idea. It's very easy to be confused to have so many different versions of our project at the same time. Even though we can communicate with teammates through e-mail time by time, SVN works efficient obviously.

Thirdly, I have to say that it's very lucky to have so excellent teammates to work together. We have meeting every Monday morning to talk about our project and exchange new ideas. It actually works. And our group leader is a very responsible person. He leads us all the time and never loses heart when we meet difficulty. Again, I'm very glad to have a chance to work with these guys.

Last but not the least, one advice for the future team is that it's really important to have a plan and try you best to always follow it. It will be suffering to face to so many deadlines appearing in front of me suddenly at the end of semester.

Rongzheng Yan

This is my first time to cooperate with so many people in a project, I learned the importance of teamwork and improve the skill of communication. I also learned to do version control, black-box testing and white-box testing, which is important in developing a program but I never systematically learned before. In addition, I learned to use OCAML, a functional language. At first I felt difficulty in thinking in a different way from C++ or JAVA; however, when we became familiar with OCAML, we found that it takes less time to debug an OCAML program. I also find that deep thinking before coding is helpful to develop a program. It is often the case that we begin in a hurry, meet a bottleneck in the process and then spend a lot of time to find the drawback in our design. If we can think carefully, it will be more efficient.

Yu Shao

The class really brought me a totally new world of programming both logic and functional. As an EE student, I used to do programming in C/C++, but I never had a chance to think out how computer languages work. I am really curious about how the code can be translated in to something that computer can understand. That drove me to challenge myself and gain the fundamental knowledge about computer languages.

At the very beginning of the project, I did no get used to Object Caml language style. I think it would be better that we can use some other development environment instead of it. However, I gradually find out functional programming language such as O'caml is a good way to design a compiler. It requires fewer lines of code to finish the project than any other tools. It is important to set up the syntax and semantic rules of our language as early as possible. As a team, we started off struggling a little but we were able to share ideas and went through the bottleneck thanks to teamwork. I really appreciate the effort and time contributed by the rest of team. It is almost impossible to complete the project with out cooperation.

Also, we got suggestions and generous help from professor Edwards and TA Hemanth. When we do retrospect, we think it is really hard to follow the timeline strictly considered using a language none of us was familiar with. Fortunately, we effectively met most of the deadlines we set at first and we complete the project on time. After working on this project I found it much easier to understand the syntax and semantic rules of program. Along with knowledge of building a compiler, the project also exposed me to makefile, shell script and several useful tools for sharing and distribution.

Yigang Zhang

I totally agree the saying that "Never had I spend so much time to write so little that does so much"!
Start early. We benefit a lot from starting early. We delivered our scanner, parser and ast very early, so we have more time work on the back end.
Team work. We all have very stressed schedule this semester. But we have regular meetings and work every week with some delivers.
Understand the problem to solve. We are really confused how to use byte code in our language at the beginning. Thanks for TA Hemanth's help.
Version control. We have some version disorders due to the bad management of source files. Latter we use google code SVN, our problem solved a lot.
Understand Microc is key to our project development.

William Chan

I learned a great deal from this project and from the class in general. This project gave me a chance to peer "behind the curtain" to see how a compiler actually works, how it is able to take an inert textfile and convert it into a program that does something useful on a computer.

I used to take it for granted that code I write just "magically" becomes a program. I came away amazed at the process and realized how much work is really involved in getting a compiler to accomplish this.

I also had a chance to learn how to use new programming tools, including programming in OCaml and using Google Docs and Google Code. There's a steep curve in learning OCaml and it's been frustrating at times, but I've learned to appreciate some of the things that OCaml can do that a C-style language doesn't do very well. As for Google Docs and Google Code, I have never used these tools prior to this project. Both turned out to be extremely useful in keeping track of our files and source code. We were able to store previous versions of everything we had, and because of that, we were able to backtrack when we messed something up. It was also extremely helpful in keeping track of all the various code segments we were working on.

Appendix

Scanner.mll

```
{
  open Parser
    (*These code is questionable, For now just leave them here.
        let incr_lineno lexbuf =
            let pos = lexbuf.Lexing.lex_curr_p in
                  lexbuf.Lexing.lex_curr_p <- { pos with
                        Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
                        Lexing.pos_bol = pos.Lexing.pos_cnum;
                  }
      *)
}
rule token = parse
    [' ' '\t' '\n' '\r'] {token lexbuf} (*White spaces*)
| "/*" {comment lexbuf}
(* Groupinp *)
| "begin"       {LBRACE}
| "end"         {RBRACE}
| '('           {LPAREN}
| ')'           {RPAREN}
| '['      {LBRACK}
| ']'      {RBRACK}
| "[|"          {LARRAY}
| "|]"          {RARRAY}
(* Operators: *)
    (* Arithmetic: *)
| '+'           {PLUS}
```

```
| '-'        {MINUS}
| '*'        {TIMES}
| '/'            {DIVIDE}
| "++"       {INCR}
| "--"       {DECR}
   (* Comparison: *)
| '>'        {GT}
| '<'        {LT}
| '='        {EQ}
| "!="       {NOTEQ}
| ">="       {GTEQ}
| "<="       {LTEQ}
   (* Logical: *)
| "not"      {NOT}
| "and"      {AND}
| "or"       {OR}
   (* Other: *)
| '.'        {DOT}
| ','        {COMMA}
| ';'        {SEMI}
| '~'        {ASSIGN}

(* scope *)
(* Types *)
| "int"      {INT}
| "bool"     {BOOLEAN}
| "string" {STRING}
| "Array"    {ARRAY}
| "Grid"     {GRID}
```

| "Cell" {CELL}

| "Row" {ROW}

| "CheckGrid" {CHECKGRID}

| "Random" {RANDOM}

| "Column" {COLUMN}

| "Diagnal" {DIAGNAL}

| "Block" {BLOCK}

| "Stack" {STACK} (*3 vertically connected blocks*)

| "Band" {BAND} (*3 horizontally connected blocks*)

(* Keywords: *)

| "if" {IF}

| "else" {ELSE}

| "for" {FOR}

| "while" {WHILE}

| "true" {TRUE(true)}

| "false" {FALSE(false)}

| "return" {RETURN}

(*give a sample sentence*)

| "MakeVisible" {VISIBLE} (*MakeVisible*)

(*| "IsRepeatable" {REPEATABLE} *)(*Whether the values in the specified cells can be duplicates.*)

(*| "DefineBlock" {DEFINEBLOCK} *) (*Allows the programmer to define a sub-square of the grid. Its row and column dimensions must be factors of the grid's rows and columns dimensions respectively.*)

(*| "SetAsGiven" {SETASGIVEN}*)

(*give a sample sentence*)

| "SumOfRows" {SUMROW} (*The total that all the values in each row

```
must sum to.*)
| "SumOfColumns"          {SUMCOL}        (*The total that all the values in each column must sum to.*)
| "SumOfDiagnals"         {SUMDIAG}       (*The total that all the values in each diagonal must sum to.*)
| "MaxValueOfRows"              {MAXROW}        (*What the maximum value of each row is permitted to be.*)
| "MaxValueOfColumns"             {MAXCOL}        (*What the maximum value of each column is permitted to be.*)
| "MaxValueOfDiagnals"            {MAXDIAG}       (*What the maximum value of each diagonal is permitted to be.*)
| "MinValueOfRows"              {MINROW}        (*Do we still need this? what the minimum value of each row is permitted to be.*)
| "MinValueOfColumns"             {MINCOL}        (*Do we still need this? what the minimum value of each row is permitted to be.*)
| "MinValueOfDiagnals"            {MINDIAG}       (*Do we still need this? what the minimum value of each diagonal is permitted to be.*)
| ['0'-'9']+ as lit {LITERAL(int_of_string lit)}
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lit { ID(lit) }
| '\"'[^ '\"']*'\"' as


    lxm { STRINGLITERAL(String.sub lxm 1 ((String.length lxm)-2)) }
| eof          {EOF}                              (*End of file*)
| _ as char { raise (Failure("illegal character "^Char.escaped char))}
| _ as char {
                    let pos = lexbuf.Lexing.lex_curr_p in
                          raise (Failure("Illegal character: " ^ Char.escaped char
                          ^ " in line #" ^ (string_of_int pos.Lexing.pos_lnum))) }
and comment = parse
   "*/" {token lexbuf}
```

| _ {comment lexbuf}

Parser.mly

```
%{ open Ast %}
%token SEMI LBRACE RBRACE LPAREN RPAREN COMMA LBRACK RBRACK LARRAY RARRAY
%token TIMES DIVIDE INCR DECR RANDOM
%token PLUS MINUS
%token GT LT EQ NOTEQ GTEQ LTEQ
%token NOT AND OR
%token DOT ASSIGN
%token INT BOOLEAN STRING GRID ARRAY
%token IF ELSE WHILE FOR RETURN
%token VISIBLE REPEATABLE
%token CELL ROW COLUMN DIAGNAL STACK BAND BLOCK
%token SUMROW SUMCOL SUMDIAG MAXROW MAXCOL MAXDIAG MINROW MINCOL MINDIAG ISVISIBLE CHECKGRID
%token LEFT RIGHT
%token <bool> TRUE FALSE
%token <int> LITERAL
%token <string> STRINGLITERAL
%token <string> ID
%token EOF
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%nonassoc NOT
%left AND OR
%left EQ NOTEQ
```

```
%left LT GT LTEQ GTEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left INCR DECR
%left BOOLEAN INT STRING GRID ARRAY
%start program
%type <Ast.program> program
%%
program:
    /* nothing*/ { [],[] }
    | program vdecl { ($2::fst $1), snd $1 }
    | program fdecl { fst $1, ($2 :: snd $1) }

vdecl:
INT ID SEMI    { {vtype=Integer;vname=$2} }
| STRING ID SEMI { {vtype=Strings;vname=$2} }
| BOOLEAN ID SEMI   { {vtype=Boolean;vname=$2} }
| ARRAY ID SEMI  { {vtype=Arrays;vname=$2} }

fdecl:
    ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
        {{ fname=$1;
          formals=$3;
          locals=List.rev $6;
          body=List.rev $7}}
```

Ast.mli

```
(* Simple binary operators *)
type op =
Add | Sub | Mult | Div | Equal | Neq | Less |
Leq | Greater | Geq | And | Or

(*type scope=
Global | Local
*)

type t=
Integer|Strings|Boolean|Arrays

type expr =
      Id of string
    |Literal of int
    |Bool of bool
    |String of string
    |Array of int list
    |Array1 of string * expr

(*   |Array2 of string * expr * expr*)
  | Binop of expr * op * expr
  | Not of expr
  | Assign of expr * expr
  | Cell of expr * expr
  | Row of expr
```

```
   | Column of expr
   | Random of expr
  (* | SetAsGiven of expr * expr * expr
   | IsRepeatable of varexp * expr*)
   | IsVisible of expr * expr
(*    | DefineBlock of expr * expr * expr * expr*)
   | SumRow of expr
   | SumCol of expr
   | SumDiag of expr
   | MaxRow of expr
   | MaxCol of expr
   | MaxDiag of expr
   | MinRow of expr
   | MinCol of expr
   | MinDiag of expr
   | CheckGrid
   | Call of string * expr list
   | Noexpr
(*type decl=
      Declare of string *t
*)
type decl={
  vtype: t;
  vname: string;
}
(*type formal=Formal of string * t*)

type formal={
ftype: t;
foname: string;
```

```
}

type stmt =
(*    Break*) (* Should break out of the current for/ while loop*)

   | IfElse of expr * stmt * stmt
   | While of expr * stmt
   | For of expr * expr* expr * stmt
   | Expr of expr
   | Return of expr
   | Block of stmt list
   | Nostmt

type func_decl = {
   fname : string ;
   formals : formal list ;
   locals :    decl list ;
   body      :    stmt list      ;
   }

type program = decl list * func_decl list (* starting point *)
let rec string_of_array=function
          []->""
          | [b]->string_of_int b
          | hd::tl->string_of_int hd^", "^string_of_array tl

let rec string_of_expr = function
      Literal(l) -> string_of_int l
      | Bool(true) -> "true"
```

```
      | Bool(false) -> "false"
      | String(s) ->"\"" ^ s ^"\""
      | Array(a) -> "[| "^
           string_of_array a
           ^" |]"
      | Array1(a,e)-> a^"[ "^string_of_expr e^" ]"
(*    | Array2(a,e1,e2) ->a^"[ "^string_of_expr e1^", "^string_of_expr e2^" ]"*)
      | Id(b)-> b
      | Binop(e1, o, e2) ->
          string_of_expr e1^" "^
           (match o with
           Add->"+" | Sub -> "-"| Mult -> "*" | Div -> "/"
           | Equal -> "=" | Neq -> "!=" | Greater -> ">"
           | Less -> "<" | Leq -> "<=" | Geq -> ">=" | And -> "and" | Or -> "or")^" "
           ^string_of_expr e2
      | Assign(e1,e2)->string_of_expr e1^"~"^string_of_expr e2
      | Not(e1)->"not "^string_of_expr e1
      | Cell(e1, e2)->"Grid.Cell("^string_of_expr e1^", "^string_of_expr e2^")"
(*    | Row(e)->"Grid.Row("^string_of_expr e^")"
      | Column(e)->"Grid.Column("^string_of_expr e^")"
      | Diagnal_left->"Grid.Diagnal(left)"
      | Diagnal_right->"Grid.Diagnal(right)"
      | Block_grid(e1,e2)->"Grid.Block("^string_of_expr e1^", "^string_of_expr e2^") "
      | Band(e)->"Grid.Band("^string_of_expr e^")"
      | Stack(e)->"Grid.Stack("^string_of_expr e^")"*)
(*    | SetAsGiven(var, e1, e2, e3)->string_of_var var^".Cell("^string_of_expr e1^",
"^string_of_expr e2^").SetAsGiven("^string_of_expr e3^")"
      | IsRepeatable(var,e)->string_of_var var^".IsRepeatable("^string_of_expr e^")"*)
      | IsVisible(e1,e2)->"Grid.IsVisible("^string_of_expr e1^", "^string_of_expr e2^")"
      | SumRow(e)->"Grid.SumofRows("^string_of_expr e^")"
```

```
      | SumCol(e)->"Grid.SumOfColumns("^string_of_expr e^")"
      | SumDiag(e)->"Grid.SumOfDiagnals("^string_of_expr e^")"
      | MaxRow(e)->"Grid.MaxValueOfRows("^string_of_expr e^")"
      | MaxCol(e)->"Grid.MaxValueOfColumns("^string_of_expr e^")"
      | MaxDiag(e)->"Grid.MaxValueOfDiagnals("^string_of_expr e^")"
      | MinRow(e)->"Grid.MinValueOfRows("^string_of_expr e^")"
      | MinCol(e)->"Grid.MinValueOfColumns("^string_of_expr e^")"
      | MinDiag(e)->"Grid.MinValueOfDiagnals("^string_of_expr e^")"
      | CheckGrid -> "Grid.CheckGrid()"
      | Call(f,e1) -> f^"( "^String.concat ", " (List.map string_of_expr e1)^" )"
      | Noexpr -> ""

let rec string_of_stmt = function
       Block(stmts) -> "begin\n"^String.concat "" (List.map string_of_stmt
stmts)^"end\n"
      | Expr(expr) -> string_of_expr expr^";\n";
      | Return(expr) -> "return " ^string_of_expr expr^";\n";
      | IfElse(expr, stmt, Block([])) -> "if( "^string_of_expr expr^" )\n"^string_of_stmt
stmt
      | IfElse(expr,stmt1,stmt2) -> "if( "^string_of_expr expr^" )\n"^string_of_stmt
stmt1^
          "else\n"^string_of_stmt stmt2
      | For(e1,e2,e3,stmt) -> "for( "^string_of_expr e1^"; "^string_of_expr e2^";
"^string_of_expr e3^" )\n"^string_of_stmt stmt
      | While(expr,stmt) -> "while( "^string_of_expr expr^" )\n"^string_of_stmt stmt
      | Nostmt->""

let string_of_t = function
      Integer->"int"
```

```
        | Strings -> "string"
        | Boolean -> "bool"
        | Arrays -> "Array"

let string_of_vdecl vdecl=string_of_t vdecl.vtype ^ " " ^ vdecl.vname^";\n"
let string_of_formal formal=string_of_t formal.ftype ^ " " ^ formal.foname

(*let string_of_formal=function
        Formal(id, Integer)->"int "^id
        | Formal(id, Strings)->"string "^id
        | Formal(id, Boolean)->"bool "^id
        | Formal(id, Arrays)->"Array "^id
          *)

let string_of_fdecl fdecl=
        fdecl.fname^" ("^String.concat ", " (List.map string_of_formal fdecl.formals)^
" )\nbegin\n"^String.concat "" (List.map string_of_vdecl fdecl.locals)^String.concat ""
(List.map string_of_stmt fdecl.body)^"end\n"
let string_of_program (vars, funcs)=

        String.concat "" (List.map string_of_vdecl vars)^"\n"^
        String.concat "\n" (List.map string_of_fdecl funcs)
```

Bytecode.ml

```
type bstmt =
    Lit of int       (* Push a literal *)
  | Lits of string
  | Lita of int list
  | Nota                    (*not *)
  | Lodg                    (*load a grid or a cell of grid*)
  | Strg of bool      (*store grid or cell with properties of whether it is visible*)
  | Srow                    (*store a row of grid object*)
  | Scol                    (*store a column of grid object*)
  | Grow   (*get a row from grid object*)
  | Gcol      (*get a column from grid object*)
  | Chkg     (*check rules set up by programmer*)
  | Setv
  | Ran          (*get a random number*)
  | Drp                (* Discard a value *)
  | Bin of Ast.op (* Perform arithmetic on top of stack *)
  | Lod of int     (* Fetch global variable *)
  | Str of int     (* Store global variable *)
  | Lfp of int     (* Load frame pointer relative *)
  | Sfp of int     (* Store frame pointer relative *)
  | Jsr of int       (* Call function by absolute address *)
  | Ent of int       (* Push FP, FP -> SP, SP += i *)
  | Rts of int       (* Restore FP, SP, consume formals, push result *)
  | Beq of int       (* Branch relative if top-of-stack is zero *)
  | Bne of int       (* Branch relative if top-of-stack is non-zero *)
  | Bra of int       (* Branch relative *)
  | Hlt                 (* Terminate *)
```

```
type prog = {
    num_globals : int;      (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }

let string_of_stmt = function
    Lit(i) -> "Lit " ^ string_of_int i
  | Lits(ls) -> "Lits \""^ ls ^"\""
  | Lita(a) -> "Lita [|"^String.concat ", " (List.map string_of_int a) ^ "|]"
  | Drp -> "Drp"
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Bin(Ast.Or) -> "Or"
  | Bin(Ast.And) -> "And"
  | Nota -> "Nota"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Lodg -> "Lodg"
  | Strg(i) -> "Strg "^(if i=true then "true" else "false")
```

```
  | Chkg -> "Chkg"
  | Setv -> "Setv"
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rt " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Hlt     -> "Hlt"

let string_of_prog p =

  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

Compile.ml

```
open Ast

open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)

type env = {

    function_index : int StringMap.t; (* Index for each function *)

    global_index   : int StringMap.t; (* "Address" for global variables *)

    local_index    : int StringMap.t; (* FP offset for args, locals *)

  }

(* val enum : int -> 'a list -> (int * 'a) list *)

let rec enum stride n = function

    [] -> []

  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
```

```
let string_map_pairs map pairs =

   List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(** Translate a program in AST form into a bytecode program.   Throw an
      exception if something is wrong, e.g., a reference to an unknown
      variable or function *)

let translate (globals, functions) =
   (* Allocate "addresses" for each global variable *)
   let global_indexes = string_map_pairs StringMap.empty (enum 1 0 (List.map (fun f
-> f.vname)globals)) in



   (* Assign indexes to function names; built-in "print" is special *)
   let add_function map (a, b)=StringMap.add a b map in
   let built_in_functions = List.fold_left add_function StringMap.empty
[("print",-1);("random",-2);("printg",-3)] in
   let function_indexes = string_map_pairs built_in_functions
         (enum 1 1 (List.map (fun f -> f.fname) functions)) in
   (* Translate a function in AST form into a list of bytecode statements *)

   let translate env fdecl =
      (* Bookkeeping: FP offsets for locals and arguments *)
      let num_formals = List.length fdecl.formals

      and num_locals = List.length fdecl.locals
      and local_offsets = enum 1 1 (List.map (fun f -> f.vname) fdecl.locals)
      and formal_offsets = enum (-1) (-2) (List.map (fun f -> f.foname) fdecl.formals) in
      let env = { env with local_index = string_map_pairs
```

```
            StringMap.empty (local_offsets @ formal_offsets) } in
   let rec expr = function
  Literal i -> [Lit i]
     | String j -> [Lits j]
     | Bool i -> if i=true then [Lit 1] else [Lit 0]
     | Array a -> [Lita a]
     | Array1 (a,e) -> expr e @ (try [Lfp (StringMap.find a env.local_index)]
               with Not_found -> try [Lod (StringMap.find a env.global_index)]
               with Not_found -> raise (Failure ("undeclared variable "^a)))
 (*     | Array2 (a,e1,e2)-> *)
     | Id s ->
   [Lit (-1)] @ (try [Lfp (StringMap.find s env.local_index)]
           with Not_found ->    try [Lod (StringMap.find s env.global_index)]
           with Not_found -> raise (Failure ("undeclared variable " ^ s)))
     | Cell(e1,e2)-> expr e1 @ expr e2 @ [Lodg]
     | Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]
     | Assign (s, e) -> (expr e) @ (match s with
      Id s1 -> [Lit (-1)] @ (try [Sfp (StringMap.find s1 env.local_index)]
    with Not_found -> try [Str (StringMap.find s1 env.global_index)]
    with Not_found -> raise (Failure ("undeclared variable " ^ s1)))
      | Array1 (a,e)->(expr e) @(try [Sfp (StringMap.find a env.local_index)]
    with Not_found -> try [Str (StringMap.find a env.global_index)]
    with Not_found -> raise (Failure ("undeclared variable " ^ a)))
      | Cell(e1,e2)->((expr e1) @ (expr e2) @ [Strg true])
(*row*)          | Row e      -> expr e @ [Srow]

                 | Column e -> expr e @ [Scol]
      | _ -> raise (Failure("unreasonable expression")))
     | Row e -> expr e @ [Grow]
     | Column e -> expr e @ [Gcol]
```

```
    | Random e -> expr e @ [Ran]
    | Not e -> expr e @ [Nota]
    | SumRow e -> expr e @ [Lit 1; Lit (-29); Strg true]
    | SumCol e -> expr e @ [Lit 1; Lit (-28); Strg true]
    | SumDiag e -> expr e @ [Lit 1; Lit (-27); Strg true]
    | MaxRow e -> expr e @ [Lit 1; Lit (-26); Strg true]
    | MaxCol e -> expr e @ [Lit 1; Lit (-25); Strg true]
    | MaxDiag e -> expr e @ [Lit 1; Lit (-24); Strg true]
    | MinRow e -> expr e @ [Lit 1; Lit (-23); Strg true]
    | MinCol e -> expr e @ [Lit 1; Lit (-22); Strg true]
    | MinDiag e -> expr e @ [Lit 1; Lit (-21); Strg true]
    | IsVisible (e1, e2) -> expr e1 @ expr e2 @ [Setv]
    | CheckGrid -> [Chkg]
    | Call (fname, actuals) -> (try
   (List.concat (List.map expr (List.rev actuals))) @
   [Jsr (StringMap.find fname env.function_index) ]
       with Not_found -> raise (Failure ("undefined function " ^ fname)))
    | Noexpr -> []


 in let rec stmt = function
Block sl         ->    List.concat (List.map stmt sl)
    | Expr e            -> expr e @ [Drp]
    | Return e         -> expr e @ [Rts num_formals]
    | IfElse (p, t, f) -> let t' = stmt t and f' = stmt f in
expr p @ [Beq(2 + List.length t')] @
t' @ [Bra(1 + List.length f')] @ f'
    | For (e1, e2, e3, b) ->
   stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))
    | While (e, b) ->
```

```
      let b' = stmt b and e' = expr e in
      [Bra (1+ List.length b')] @ b' @ e' @
      [Bne (-(List.length b' + List.length e'))]
       | Nostmt -> []

    in [Ent num_locals] @          (* Entry: allocate space for locals *)
    stmt (Block fdecl.body) @    (* Body *)
    [Lit 0; Rts num_formals]      (* Default = return 0 *)

in let env = { function_index = function_indexes;
          global_index = global_indexes;
          local_index = StringMap.empty } in

(* Code executed to start the program: Jsr main; halt *)
let entry_function = try
    [Jsr (StringMap.find "main" function_indexes); Hlt]
with Not_found -> raise (Failure ("no \"main\" function"))
in

(* Compile the functions *)
let func_bodies = entry_function :: List.map (translate env) functions in
(* Calculate function entry points by adding their lengths *)
let (fun_offset_list, _) = List.fold_left
      (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in

let func_offset = Array.of_list (List.rev fun_offset_list) in
{ num_globals = List.length globals;

    (* Concatenate the compiled functions and replace the function
         indexes in Jsr statements with PC values *)
```

```
    text = Array.of_list (List.map (function
   Jsr i when i > 0 -> Jsr func_offset.(i)
     | _ as s -> s) (List.concat func_bodies))
 }
```

Execute.ml

```
open Ast
open Bytecode
let execute_prog prog =
  let stack = Array.make 1024 ["0"]
  and globals = Array.make prog.num_globals ["0"]
  and grid = Array.make 111 (0,false)
  in
  let rec exec fp sp pc = match prog.text.(pc) with
      Lit i    -> stack.(sp) <- [string_of_int i] ; exec fp (sp+1) (pc+1)
  | Lits j -> stack.(sp) <- [j] ; exec fp (sp+1) (pc+1)
  | Lita a -> stack.(sp) <- List.map string_of_int a ; exec fp (sp+1) (pc+1)
  | Drp       -> exec fp (sp-1) (pc+1)
  | Bin op -> let op1 = (if ( int_of_string (List.hd stack.(sp-2))=(-1) ) then ( int_of_string
(List.hd stack.(sp-3)) )
                                else ( int_of_string (List.hd stack.(sp-2)) ) )
               and op2 =int_of_string (List.hd    stack.(sp-1)) in
               stack.(sp-2) <- [string_of_int (let boolean i = if i then 1 else 0 in
                     match op with
         Add         -> op1 + op2
```

```
| Sub      -> op1 - op2
| Mult     -> op1 * op2
| Div      -> op1 / op2
| Or       -> let a=(if op1=0 then false else true) and b=(if op2=0 then false else
true) in boolean (a || b)
| And      -> let a=(if op1=0 then false else true) and b=(if op2=0 then false else
true) in boolean (a && b)

| Equal    -> boolean (op1 =  op2)
| Neq      -> boolean (op1 != op2)
| Less     -> boolean (op1 <  op2)
| Leq      -> boolean (op1 <= op2)
| Greater -> boolean (op1 >  op2)
| Geq      -> boolean (op1 >= op2))] ;
exec fp (sp-1) (pc+1)
| Nota -> if int_of_string (List.hd stack.(sp-1))=0 then stack.(sp-1) <- ["1"]
  else stack.(sp-1) <- ["0"] ; exec fp sp (pc+1)
(*row*)
  | Grow   ->(*stack.(sp)*) let t= let row = int_of_string(List.hd(stack.(sp-1))) in
                      (let rec range a b =
                         if b > 8 then []
                         else    string_of_int(fst(grid.(a)))::range   (a+1)
(b+1)
                      in range (30+9*(row-1)) 0 ) in stack.(sp)<-t;
                      exec fp (sp+1) (pc+1)
  | Srow   -> let row = int_of_string(List.hd(stack.(sp-1))) in
              let rec store n = function
                    []  ->  ()
                  |  hd::tl  ->  grid.((row-1)*9+30+n)  <-  (int_of_string(hd),true);
store (n+1) tl
```

```
                    in store 0 (stack.(sp-2)) ; exec fp (sp+1) (pc+1)
(*column*)
   | Gcol     -> let t = let col = int_of_string(List.hd(stack.(sp-1)))in
                    (let rec range a b =
                          if b < 9 then string_of_int(fst(grid.(a)))::range (a+9) (b+1)
                          else    []
                    in range ((col-1)+30) 0) in stack.(sp) <- t; exec fp (sp+1) (pc+1)

   | Scol     -> let col = int_of_string(List.hd(stack.(sp-1))) in
                    let rec store n = function
                             []    -> ()
                          | hd::tl -> grid.((col-1)+30+n*9) <- (int_of_string(hd),true); store
(n+1) tl
                    in store 0 (stack.(sp-2))    ; exec fp (sp+1) (pc+1)
      | Setv -> let row = int_of_string (List.hd stack.(sp-2)) and col=int_of_string (List.hd
stack.(sp-1)) in
                    let num = fst (grid.(30+9*row+col)) in
grid.(30+9*row+col)<-(num,true); exec fp sp (pc+1)
  (* | Lod i     -> stack.(sp)     <- (Array.of_list (globals.(i))) .(stack(sp-1))   ; exec fp
(sp+1) (pc+1)*)
      | Lod i -> if int_of_string (List.hd stack.(sp-1)) = -1 then stack.(sp) <- globals.(i)
          else stack.(sp) <- [List.nth globals.(i) ((int_of_string (List.hd stack.(sp-1)))-1)] ;
exec fp (sp+1) (pc+1)
      | Str i     -> if List.hd stack.(sp-1) = "-1" then globals.(i) <- stack.(sp-2)
          else (let atemp=Array.of_list globals.(i) in
                     atemp.(int_of_string (List.hd stack.(sp-1))-1)<-List.hd stack.(sp-3);
                     let ltemp=Array.to_list atemp in   globals.(i)<- ltemp) ; exec fp sp
(pc+1)
  (* | Lfp i     -> stack.(sp)     <- Array.of_list(stack.(fp+i)).(stack(sp-1)) ; exec fp (sp+1)
(pc+1)*)
```

```
| Lfp i -> if int_of_string (List.hd stack.(sp-1))= -1 then stack.(sp) <- stack.(fp+i)
    else stack.(sp) <- [List.nth stack.(fp+i) (int_of_string (List.hd stack.(sp-1)))] ;
exec fp (sp+1) (pc+1)
| Sfp i   ->  if List.hd stack.(sp-1) = "-1" then stack.(fp+i) <- stack.(sp-2)
    else (let atemp=Array.of_list (stack.(fp+i)) in
            atemp.(int_of_string (List.hd stack.(sp-1))-1)<-(List.hd stack.(sp-3));
            let ltemp=Array.to_list atemp in stack.(fp+i)<- ltemp)  ; exec fp sp
(pc+1)
 | Lodg    ->  if List.hd stack.(sp-2) = "-1" then stack.(sp) <- [string_of_int (fst
(grid.(30+9*(int_of_string (List.hd stack.(sp-3))-1)+(int_of_string (List.hd
stack.(sp-1))-1))))]
          else stack.(sp) <- [string_of_int (fst (grid.(30+9*((int_of_string (List.hd
stack.(sp-2)))-1)+(int_of_string (List.hd stack.(sp-1)))-1)))]; exec fp (sp+1) (pc+1)
 | Strg i  ->  if (List.hd stack.(sp-2) = "-1") then grid.(30+9*(int_of_string
(List.hd stack.(sp-3))-1)+(int_of_string (List.hd stack.(sp-1)))-1)<- if (List.hd stack.(sp-4)
= "-1") then (int_of_string (List.hd stack.(sp-5)), i)
                    else (int_of_string (List.hd stack.(sp-4)), i)
          else (if (List.hd stack.(sp-3) = "-1") then grid.(30+9*((int_of_string
(List.hd stack.(sp-2))-1))+(int_of_string (List.hd stack.(sp-1))-1))<-(int_of_string
(List.hd stack.(sp-4)), i)
          else grid.(30+9*((int_of_string (List.hd stack.(sp-2))-1))+(int_of_string
(List.hd stack.(sp-1))-1))<-(int_of_string (List.hd stack.(sp-3)), i)) ;exec fp sp (pc+1)
 | Jsr(-1) ->  if List.length (stack.(sp-1))=1
          then print_endline (List.hd stack.(sp-1))
          else print_endline("[|" ^ String.concat "," stack.(sp-1) ^ "|]") ;
  exec fp sp (pc+1)
 | Jsr(-2) ->  stack.(sp) <- [string_of_int (Random.self_init ();Random.int 9)] ; exec
fp (sp+1) (pc+1)
 | Jsr(-3) ->  for i=0 to 8 do
     for j=0 to 8 do
```

```
        if snd (grid.(30+i*9+j))=true then print_string (string_of_int(fst grid.(30+i*9+j))^" ")(*print_int (fst grid.(30+i*9+j))*)
        else print_string "* "(*print_int (fst grid.(30+i*9+j))*)
      done;
      print_endline " "
        done ; exec fp sp (pc+1)
    | Ran     ->  stack.(sp) <- [string_of_int (Random.self_init (); Random.int ( int_of_string (List.hd stack.(sp-1)))+1)]; exec fp (sp+1) (pc+1)
  | Chkg    ->  if snd grid.(0)=true then for i=0 to 8 do
                          let rec sum b = if b<9 then (sum (b+1))+(fst grid.(30+i*9+b)) else 0 in let a=(sum 0) in if a != fst grid.(0) then print_endline ("Warning: Sum of row "^string_of_int (i+1)^" = " ^string_of_int a^", breaks the sum of row rule!") else ()
                  done
             else ();(*Check the sum of row rule*)
        if snd grid.(1)=true then for i=0 to 8 do
                  let rec sum b = if b<9 then (sum (b+1))+(fst grid.(30+b*9+i)) else 0 in let a=(sum 0) in if a != fst grid.(1) then print_endline ("Warning: Sum of column "^string_of_int (i+1)^" = "^string_of_int a^", breaks the sum of column rule!") else ()
                  done
             else ();(*Check the sum of column rule*)


         if snd grid.(2)=true then
                      let rec sum b = if b<9 then (sum (b+1))+(fst grid.(30+b*9+b)) else 0 in let a = (sum 0) in if a != fst grid.(2) then print_endline "Left diagnal breaks the sum of diagnal rule" else ();


                            let rec sums c = if c<9 then (sums (c+1))+(fst grid.(30+9*c+8-c)) else 0 in let b = (sums 0) in if b != fst grid.(2) then
```

```
print_endline "Right diagnal breaks the sum of diagnal rule" else()

                                        else();(*Check the sum of diagnal rule*)

            if snd grid.(3)=true then for i=0 to 8 do
                                    let rec max a b= if a<9 then (if b<(fst
grid.(30+i*9+a)) then max (a+1) (fst grid.(30+i*9+a)) else max (a+1) b) else b in let
c=max 0 0 in
                  if c > fst grid.(3) then print_endline ("Row "^string_of_int (i+1)^"
breaks the maximum of row rule") else ()
                                             done
                                          else ();(*Check the max of row rule*)

             if snd grid.(4)=true then for i=0 to 8 do
                                   let rec max a b= if a<9 then (if b<(fst
grid.(30+a*9+i)) then max (a+1) (fst grid.(30+a*9+i)) else max (a+1) b) else b in let
c=max 0 0 in
                      if c > fst grid.(4) then print_endline ("Column "^string_of_int
(i+1)^" breaks the maximum of column rule") else ()
done else (); (*Check the max of column rule*)

          if snd grid.(5)=true then let rec max a b = if a<9 then (if b<(fst
grid.(30+a*9+a)) then max (a+1) (fst grid.(30+a*9+a)) else max (a+1) b) else b in let c=
max 0 0 in if c> fst grid.(5) then print_endline "Left diagnal breaks the maximum of
diagnal rule" else ();

              let rec maxr a b = if a<9 then (if b<(fst grid.(30+a*9+8-a)) then maxr
(a+1) (fst grid.(30+a*9+8-a)) else maxr (a+1) b) else b in let c= maxr 0 0 in if c> fst
grid.(5) then print_endline "Right diagnal breaks the maximum of diagnal rule" else ()
```

```
else (); (*Check the max of diagnal rule*)

        if snd grid.(6)=true then for i=0 to 8 do
                        let rec min a b= if a<9 then (if b>(fst grid.(30+i*9+a)) then min (a+1) (fst grid.(30+i*9+a)) else min (a+1) b) else b in let c=min 0 1000 in
                    if c < fst grid.(6) then print_endline ("Row "^string_of_int (i+1)^" breaks the minimum of row rule") else ()

done else ();(*Check the min of row rule*)

        if snd grid.(7)=true then for i=0 to 8 do
                        let rec min a b= if a<9 then (if b>(fst grid.(30+a*9+i)) then min (a+1) (fst grid.(30+a*9+i)) else min (a+1) b) else b in let c=min 0 1000 in
                    if c < fst grid.(7) then print_endline ("Column "^string_of_int (i+1)^" breaks the minimum of column rule") else ()
done else (); (*Check the min of column rule*)

         if snd grid.(8)=true then let rec min a b = if a<9 then (if b>(fst grid.(30+a*9+a)) then min (a+1) (fst grid.(30+a*9+a)) else min (a+1) b) else b in let c= min 0 1000 in if c> fst grid.(8) then print_endline "Left diagnal breaks the maximum of diagnal rule" else ();
                    let rec minr a b = if a<9 then (if b>(fst grid.(30+a*9+8-a)) then minr (a+1) (fst grid.(30+a*9+8-a)) else minr (a+1) b) else b in let c= minr 0 1000 in if c> fst grid.(8) then print_endline "Right diagnal breaks the maximum of diagnal rule" else ()
else (); (*Check the min of diagnal rule*)
exec fp sp (pc+1)
  | Jsr i    -> stack.(sp)    <- [string_of_int (pc + 1)]         ; exec fp (sp+1) i
  | Ent i    -> stack.(sp)    <- [string_of_int fp]               ; exec sp (sp+i+1) (pc+1)
  | Rts i    -> let new_fp =int_of_string (List.hd stack.(fp)) and new_pc = int_of_string (List.hd stack.(fp-1)) in
```

```
                  stack.(fp-i-1) <- stack.(sp-1) ; exec new_fp (fp-i) new_pc
  | Beq i     -> exec fp (sp-1) (pc + if int_of_string (List.hd stack.(sp-1)) =   0 then i else
1)
  | Bne i     -> exec fp (sp-1) (pc + if int_of_string (List.hd stack.(sp-1)) != 0 then i else 1)
  | Bra i     -> exec fp sp (pc+i)
  | Hlt       -> ()
  in exec 0 0 0
```

(Compiling a sample program with the **Java-Ast** version):

To compile a sample program using the Java-Ast version of our compiler, send the location and name of the sample program to jsgdl.exe, for example: "./jsgdl.exe -j < ./sample-program.sgdl". The compiler will produce two files, a Grid.java file that contains the Grid class, and a Sudoku.java file that contains the main program.

Astjava.ml

```
open Ast
type jop = JAdd | JSub | JMult | JDiv | JMod | JEqual | JNeq | JLess | JLeq | JGreater
   | JGeq | JAnd | JOr

type jup = JIncr | JDecr

type jgrid = {
       jboard            : int list * int list;
       jvisible         : bool list * bool list;
       jblock            : int;
       jrepeatable    : bool;
   }

type jexpr =

     JLiteral of int

  | JStrLit of string

  | JBoolLit of bool

  | JDirectLit of bool

  | JBinop of jexpr * jop * jexpr

  | JNot of jexpr

  | JUnop of string * jup
```

```
  | JId of string
  | JAssign of string * jexpr
  | JArrLit of string * jexpr list
  | JArrayAssign of string * jexpr * jexpr list
  | JDot of string * string * jexpr list
  | JCall of string * jexpr list
  | JNoexpr

type jvdecl =
    JIntDecl of string
  | JStrDecl of string
  | JBoolDecl of string
  | JArrDecl of string * jexpr list
  | JGridDecl of string * jexpr list


type jstmt =
     JBlock of jstmt list
   | JExpr of jexpr
   | JReturn of jexpr
   | JIf of jexpr * jstmt * jstmt
   | JFor of jexpr * jexpr * jexpr * jstmt
   | JWhile of jexpr * jstmt
```

```
type jfunc_decl = {
    jfname : string;
    jformals : jvdecl list;
    jlocals : jvdecl list;
    jbody : jstmt list;
  }

type jprogram = string list * jfunc_decl list

let rec string_of_jexpr = function
    JLiteral(l) -> string_of_int l
  | JStrLit (s) -> s
  | JBoolLit (b) -> string_of_bool b
  | JDirectLit (d) -> if d then "left" else "right"
  | JBinop(e1, o, e2) ->
      string_of_jexpr e1 ^ " " ^
      (match o with
   JAdd -> "+" | JSub -> "-" | JMult -> "*" | JDiv -> "/" | JMod -> "%"
      | JEqual -> "=" | JNeq -> "!="
      | JLess -> "<" | JLeq -> "<=" | JGreater -> ">" | JGeq -> ">=" | JAnd -> "&&" |
JOr -> "||") ^ " " ^ string_of_jexpr e2
  | JNot (n) -> "!!" ^ string_of_jexpr n
  | JUnop(v, o) -> v ^ (match o with JIncr -> "++" | JDecr -> "--")
  | JId(s) -> s
```

```
  | JAssign(v, e) -> v ^ " ~ " ^ string_of_jexpr e
  | JArrLit (a, el) -> a ^ "[" ^ String.concat "][" (List.map string_of_jexpr el) ^ "]"
  | JArrayAssign (a, e, el) -> a ^ "[" ^
      String.concat "][" (List.map string_of_jexpr el) ^ "]" ^ " ~ " ^ string_of_jexpr e
  | JDot(s1, s2, el) -> s1 ^ "." ^ s2 ^ "(" ^
      String.concat ", " (List.map string_of_jexpr el) ^ ")"
  | JCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_jexpr el) ^ ")"
  | JNoexpr -> ""

let rec string_of_jstmt = function
    JBlock(stmts) ->
      "begin\n" ^ String.concat "" (List.map string_of_jstmt stmts) ^ "end\n"
  | JExpr(expr) -> string_of_jexpr expr ^ ";\n";
  | JReturn(expr) -> "return " ^ string_of_jexpr expr ^ ";\n";
  | JIf(e, s, JBlock([])) -> "if (" ^ string_of_jexpr e ^ ")\n" ^ string_of_jstmt s
  | JIf(e, s1, s2) ->    "if (" ^ string_of_jexpr e ^ ")\n" ^
      string_of_jstmt s1 ^ "else\n" ^ string_of_jstmt s2
  | JFor(e1, e2, e3, s) ->
      "for (" ^ string_of_jexpr e1    ^ " ; " ^ string_of_jexpr e2 ^ " ; " ^
      string_of_jexpr e3    ^ ") " ^ string_of_jstmt s
  | JWhile(e, s) -> "while (" ^ string_of_jexpr e ^ ") " ^ string_of_jstmt s
```

```
let string_of_jvdecl = function
    JIntDecl (id) -> "int " ^ id ^ ";\n"
  | JStrDecl (id) -> "string " ^ id ^ ";\n"
  | JBoolDecl (id) -> "bool " ^ id ^ ";\n"
  | JArrDecl (id, el) -> "array " ^ id ^    "(" ^
      String.concat ", " (List.map string_of_jexpr el) ^ ");\n"
  | JGridDecl (id, el) -> "grid " ^ id ^    "(" ^
      String.concat ", " (List.map string_of_jexpr el) ^ ");\n"

let string_of_jfdecl fdecl =
  fdecl.jfname ^ "(" ^ String.concat ", " (List.map string_of_jvdecl fdecl.jformals) ^
      ")\nbegin\n" ^
  String.concat "" (List.map string_of_jvdecl fdecl.jlocals) ^
  String.concat "" (List.map string_of_jstmt fdecl.jbody) ^
  "end\n"

let string_of_jprogram (vars, funcs) =
  String.concat "" (List.map string_of_jvdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_jfdecl funcs)
```

Jcompile.ml

```
open Ast

open Astjava

let java_of_program (globals, functions) =

let rec java_of_expr = function
      Literal(l) -> JLiteral(l)

   | StrLit (s) -> JStrLit(s)

   | BoolLit (b) -> JBoolLit(b)

   | DirectLit (d) -> JDirectLit(d)

   | Binop(e1, o, e2) ->

         (match o with

     Add ->        JBinop (java_of_expr e1, JAdd, java_of_expr e2)

     | Sub ->       JBinop (java_of_expr e1, JSub, java_of_expr e2)

     | Mult ->      JBinop (java_of_expr e1, JMult, java_of_expr e2)

     | Div ->       JBinop (java_of_expr e1, JDiv, java_of_expr e2)

     | Mod ->       JBinop (java_of_expr e1, JMod, java_of_expr e2)
```

```
      | Equal ->      JBinop (java_of_expr e1, JEqual, java_of_expr e2)

      | Neq ->        JBinop (java_of_expr e1, JNeq, java_of_expr e2)

      | Less ->       JBinop (java_of_expr e1, JLess, java_of_expr e2)

      | Leq ->        JBinop (java_of_expr e1, JLeq, java_of_expr e2)

      | Greater -> JBinop (java_of_expr e1, JGreater, java_of_expr e2)

      | Geq ->        JBinop (java_of_expr e1, JGeq, java_of_expr e2)

      | And ->        JBinop (java_of_expr e1, JAnd, java_of_expr e2)

      | Or ->         JBinop (java_of_expr e1, JOr, java_of_expr e2))

      | Not (n) -> JNot(java_of_expr n)

      | Unop(v, o) -> (match o with Incr -> JUnop (v, JIncr) | Decr -> JUnop (v, JDecr))

    | Id(s) -> JId(s)

    | Assign(v, e) -> JAssign(v, java_of_expr e)

    | ArrLit (a, el) -> JArrLit(a, List.map java_of_expr el)

    | ArrayAssign (a, e, el) -> JArrayAssign (a, java_of_expr e, List.map java_of_expr el)

    | Dot(s1, s2, el) -> JDot (s1, s2, List.map java_of_expr el)

    | Call(f, el) -> JCall (f, List.map java_of_expr el)

    | Noexpr -> JNoexpr


in let rec java_of_stmt = function

        Block(stmts) -> JBlock (List.map java_of_stmt stmts)

    | Expr(expr) -> JExpr(java_of_expr expr)

    | Return(expr) -> JReturn (java_of_expr expr)
```

```
   | If(e, s, Block([])) -> JIf(java_of_expr e, java_of_stmt s, JBlock([]))

   | If(e, s1, s2) ->    JIf(java_of_expr e, java_of_stmt s1, java_of_stmt s2)

   | For(e1, e2, e3, s) ->

          JFor(java_of_expr e1, java_of_expr e2, java_of_expr e3, java_of_stmt s)

   | While(e, s) -> JWhile(java_of_expr e, java_of_stmt s)


in let java_of_vdecl = function

      IntDecl (id) -> JIntDecl (id)

   | StrDecl (id) -> JStrDecl (id)

   | BoolDecl (id) -> JBoolDecl (id)

   | ArrDecl (id, el) -> JArrDecl (id, List.map java_of_expr el)

   | GridDecl (id, el) -> JGridDecl (id, List.map java_of_expr el)


in let java_of_fdecl fdecl =

    let jfdecl = {

       jfname = fdecl.fname;

       jformals = List.map java_of_vdecl fdecl.formals;

       jlocals = List.map java_of_vdecl fdecl.locals;

       jbody = List.map java_of_stmt fdecl.body;    }

   in jfdecl

in (List.map java_of_vdecl globals, List.map java_of_fdecl functions)
```

Jexecute.ml

```
open Astjava

let exec_jprogram (vars, funcs) =
let gridclass = "
public class Grid
{
	protected int[][] board;
	protected boolean[][] visible;
	protected int blockSize, numRows, numCols;

	public Grid (int r, int c, int b)
	{
		board = new int[r][c];
		visible = new boolean[r][c];
		numRows = r;
```

```
        numCols = c;

        blockSize = b;

}



public boolean IsVisible ( int r, int c )

{
        return visible[r][c];

}

public void MakeVisible ( int r, int c, boolean b )

{

        visible[r][c] = b;

}



public int getCell ( int r, int c )

{

        return board[r][c];

}



public int[] getRow ( int r)

{

        int[] array = new int[numCols];

        for (int j = 0; j < numCols; j++)

        {
```

```
            array[j] = board[r][j];
        }
        return array;
    }

    public int[] getColumn ( int c )
    {
        int[] array = new int[numRows];
        for (int i = 0; i < numRows; i++)
        {
            array[i] = board[i][c];
        }
        return array;
    }

    public int[] getDiagonal ( boolean direction )
    {
        int[] array = new int[Math.min(numRows,numCols)];
        if (direction)
        {
            for (int i = 0; i < Math.min(numRows,numCols); i++)
            {
```

```
            array[i] = board[i][i];
        }
    }
    else
    {
        for (int i = 0; i < Math.min(numRows,numCols); i++)
        {
            array[i] = board[i][numCols-i];
        }
    }
    return array;
}
public int[][] getBand ( int r )
{
    int[][] array = new int[blockSize][numCols];
    for (int i = r; i < blockSize; i++)
    {
        for (int j = 0; j < numCols; j++)
        {
            array[i][j] = board[i][j];
        }
    }
    return array;
```

```
}
public int[][] getStack ( int c )
{
    int[][] array = new int[numRows][blockSize];
    for (int i = 0; i < numRows; i++)
    {
        for (int j = c; j < blockSize; j++)
        {
            array[i][j] = board[i][j];
        }
    }
    return array;
}
public int[][] getBlock ( int r, int c )
{
    int[][] array = new int[blockSize][blockSize];
    for (int i = r; i < blockSize; i++)
    {
        for (int j = c; j < blockSize; j++)
        {
            array[i][j] = board[i][j];
        }
```

```
            }

            return array;

      }



}
"and mainprog = "

import java.util.*;



public class Sudoku

{

          BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

          Random generator = new Random();

“in let rec exec_jexpr = function

        JLiteral(l) -> string_of_int l

    | JStrLit (s) -> "\"" ^ s ^"\""

    | JBoolLit (b) -> string_of_bool b

    | JDirectLit (d) -> if d then "left" else "right"

    | JBinop(e1, o, e2) ->

            exec_jexpr e1 ^ " " ^

            (match o with

       JAdd -> "+" | JSub -> "-" | JMult -> "*" | JDiv -> "/" | JMod -> "%"

            | JEqual -> "==" | JNeq -> "!="

            | JLess -> "<" | JLeq -> "<=" | JGreater -> ">" | JGeq -> ">=" | JAnd -> "&&"
```

```
        | JOr -> "||") ^ " " ^ exec_jexpr e2
    | JNot (n) -> "!" ^ exec_jexpr n
    | JUnop(v, o) -> v ^ (match o with JIncr -> "++" | JDecr -> "--")
    | JId(s) -> s
    | JAssign(v, e) -> v ^ " = " ^ exec_jexpr e
    | JArrLit (a, el) -> a ^ "[" ^ String.concat "][" (List.map exec_jexpr el) ^ "]"
    | JArrayAssign (a, e, el) -> a ^ "[" ^
        String.concat "][" (List.map exec_jexpr el) ^ "]" ^ " = " ^ exec_jexpr e
    | JDot(s1, s2, el) -> s1 ^ "." ^ s2 ^ "(" ^
        String.concat ", " (List.map exec_jexpr el) ^ ")"
    | JCall(f, el) -> (match f with
        | "print"    ->
              "System.out.println" ^ "(" ^ String.concat ", " (List.map exec_jexpr el)
^ ")"
        | "scan"      -> "in.readLine()"
        | "random" -> "generator.nextInt()"
        | _               -> f ^ "(" ^ String.concat ", " (List.map exec_jexpr el) ^ ")")
    | JNoexpr -> ""

in let rec exec_jstmt = function
      JBlock(stmts) ->
        "{\n" ^ String.concat "" (List.map exec_jstmt stmts) ^ "}\n"
```

```
    | JExpr(expr) -> exec_jexpr expr ^ ";\n";
    | JReturn(expr) -> "return " ^ exec_jexpr expr ^ ";\n";
    | JIf(e, s, JBlock([])) -> "if (" ^ exec_jexpr e ^ ")\n" ^ exec_jstmt s
    | JIf(e, s1, s2) ->    "if (" ^ exec_jexpr e ^ ")\n" ^
        exec_jstmt s1 ^ "else\n" ^ exec_jstmt s2
    | JFor(e1, e2, e3, s) ->
        "for (" ^ exec_jexpr e1    ^ " ; " ^ exec_jexpr e2 ^ " ; " ^
        exec_jexpr e3    ^ ") " ^ exec_jstmt s
    | JWhile(e, s) -> "while (" ^ exec_jexpr e ^ ") " ^ exec_jstmt s

in let exec_jvdecl = function
      JIntDecl (id) -> "int " ^ id ^ ";\n"
    | JStrDecl (id) -> "String " ^ id ^ ";\n"
    | JBoolDecl (id) -> "boolean " ^ id ^ ";\n"
    | JArrDecl (id, el) ->
        if List.length el = 1
          then "int[] " ^ id ^ " = new int[" ^
              String.concat "][" (List.map exec_jexpr el) ^ "];\n"
          else "int[][] " ^ id ^ " = new int[" ^
              String.concat "][" (List.map exec_jexpr el) ^ "];\n"
    | JGridDecl (id, el) -> "Grid " ^ id ^    " = new Grid(" ^
         String.concat ", " (List.map exec_jexpr el) ^ ");\n"
```

```
in let exec_jfdecl jfdecl = match jfdecl.jfname with
   "main" ->
        "public static void main (String[] args)\n{\n" ^
        String.concat "" (List.map exec_jvdecl jfdecl.jlocals) ^
        String.concat "" (List.map exec_jstmt jfdecl.jbody) ^ "}\n"
  | _ ->
        jfdecl.jfname ^ "(" ^
        String.concat ", " (List.map exec_jvdecl jfdecl.jformals) ^ ")\n{\n" ^
        String.concat "" (List.map exec_jvdecl jfdecl.jlocals) ^
        String.concat "" (List.map exec_jstmt jfdecl.jbody) ^ "}\n"
in let classfile = open_out "./Grid.java"
     and sudokufile = open_out "./Sudoku.java" in
        output_string classfile gridclass;
        output_string sudokufile (mainprog ^
             String.concat "" (List.map exec_jvdecl vars) ^ "\n" ^
             String.concat "\n" (List.map exec_jfdecl funcs) ^ "\n}\n");
type action = Ast | Astjava | JCompile
let _ =
   let action = if Array.length Sys.argv > 1 then
      List.assoc Sys.argv.(1) [ ("-a", Ast);
                         ("-s", Astjava);
```

```
                        ("-j", JCompile) ]
    else JCompile in
    let lexbuf = Lexing.from_channel stdin in
    let program = Parser.program Scanner.token lexbuf in
    match action with
      Ast -> let listing = Ast.string_of_program program
               in print_string listing
    | Astjava -> let listing = Astjava.string_of_jprogram (Jcompile.java_of_program program)
               in print_string listing
    | JCompile -> Jexecute.exec_jprogram (Jcompile.java_of_program program)
```