Compiler Construction Stage 1

Group 1		Group 2	
Divyanshu Goyal	2011B4A7795P	Siddharth Bhatia	2011B4A7680P
Abhishek Modi	2011B4A7331P	Akshay Aurora	2011B4A7658P

Language Description

The language is a **general purpose language** with neat and powerful constructs. It supports **structured programming**, **lexical variable scope** and **recursion** along with **dynamic type inference**. In addition the language provides **special support for String and Grid** data structures to handle character arrays and 2-dimensional integer arrays easily. The language also features a **mutable construct** which enables the programmer to declare a variable mutable/immutable i.e. allow/disallow change after first initialization (By default all variables are immutable).

Language Features

- Data Types
  - `i32, f32, char, boolean, string`
- Type inference
  - Data types can be initialised without explicitly mentioning data type
- Mutability
  - By default, all variables are immutable. To initialise variables as mutable - `mut` modifier can be used.
- Multiple assignments
  - Multiple variables can be initialised in single statement
- Strings
- Functions
  - Support for Overloading
- Conditional statements if-else
- Iterative statements
  - Support for while statements with break
- Arithmetic operators (+ - * /)
  - Defined on i32 and f32
- Logical operators (and or not)
  - Defined on boolean expressions
- Relational operators (> < = <= >= <> ==)
  - Defined for i32, f32 and strings

- **Looping**

```
while(x < 5) {
   print(x);
   x = x+1;
}
```

- **Functions**

```
fn print_number(x: i32) {
      print("x is: {}", x);
}
```

- **Conditional**

```
if (x == 5) {
    print("x is five!");
} elseif (x <> 5) {
    print("x is not five :(");
}
```

- **Arrays**

```
let a = [1, 2, 3];
let str = ['c', 'b', 'h' ];
a[2]  = 5;
```

- **Strings**

```
let mut s= "Hello there.";
s.size();
s.to_upper();
s.reverse();
```

- **Grid: 2D array structure**

```
Grid x[10][1]
x [i][j] = 10
```

- **Multiple Assignment**

```
(x,y,z) = (1,2,3);
```

- **Type Inference & Comments**

```
let x = 5; //variable x of type i32
let mut x = 6; //mutable variable
```

Tokens

Pattern	Token	Purpose
=	ASSIGNOP	Assignment operator
//[^\n]	COMMENT	Comment
[a-z|A-Z] [a-zA-Z0-9_]*	ID	Identifier (used as Variables)
[0-9][0-9]*	NUM	Integer number
[0-9][0-9]*.[0-9][0-9]	FLOAT	Real number
["][a-z][a-z]*["]	STRL	String literal
['][a-z][']	CHARL	Char literal
return	RETURN	**Keyword** return
char	CHAR	**keyword** char
i32	I32	**Keyword** int
f32	F32	**Keyword** real
bool	BOOL	**Keyword** bool
string	STRING	**Keyword** string
main	MAIN	**Keyword** main
fn	FN	**Keyword** fn
let	LET	**Keyword** let
while	WHILE	**Keyword** while
break	BREAK	**Keyword** break
[	OSQUARE	Left Square bracket
]	CSQUARE	Right Square bracket
(	OPAREN	Open parenthesis
)	CPAREN	Close parenthesis
{	OBRACE	Open braces
}	CBRACE	Close braces

;	SEMICOLON	Semicolon as separator
:	COLON	Colon
,	COMMA	Comma
if	IF	**Keyword** if
else	ELSE	**Keyword** else
elseif	ELSEIF	**Keyword** elseif
scan	SCAN	**Keyword** scan
print	PRINT	**Keyword** print
+	PLUS	Addition operator
-	MINUS	Subtraction operator
*	MUL	Multiplication operator
/	DIV	Division operator
and	AND	Logical and
or	OR	Logical or
not	NOT	Logical not
<	LT	Relational operator less than
<=	LE	Relational operator less than or equal to
==	EQ	Relational operator equal to
>	GT	Relational operator greater than
>=	GE	Relational operator greater than or equal to
<>	NE	Relational operator not equal to
->	RARROW	Return Type beginning
.	DOT	Method beginning

Grammer

Grammar = (NonTerminals, Terminals, Rules, Start)

Start = Program

NonTerminals = {Program, Functions, FunctionDef, fnReturn, Statements, moreStmts, Stmt, ReturnStmt, BreakStmt, DeclarationStmt, moreDeclarations, mutMod, Declaration, moreTypes, AssignStmtType2, listTypes, typeList, moreList, singleAssn, multAssn, moreAssn, IDStmts, IDStmts2, Index, moreIndex, AssignStmtType1, FunCall, MethodCall, FunCallStmt, MethodStmt, Type, parameterList, remainingList, IfStmt, ElseStmt, IStmt, OStmt, value, array, IDList, moreIds, arithExpn, moreTerms, arithTerm, moreFactors, factor, opLow, relType, opHigh, boolExpn, logicalOp, relationalOp, LoopStmt, grid, rows, moreRows, row, moreNums, boolean}

Terminals = {MAIN, OPAREN, CPAREN, OBRACE, CBRACE, FN, ID, COMMENT, RETURN, SEMICOLON, Break, LET, MUT, ASSIGNOP, COMMA, COLON, OSQUARE, CSQUARE, NUM, DOT, BOOL, F32, I32, string, type, IF, ELSEIF, ELSE, SCAN, PRINT, CHARL, STRINGL, FLOAT, MINUS, PLUS, DIV, MUL, AND, NOT, OR, EQ, GT, GTE, LT, LTE, NE, WHILE, FALSE, TRUE}

Rules = {

Program	→ Functions MAIN OPAREN CPAREN OBRACE Statements CBRACE
Functions	→ FunctionDef Functions | ε
FunctionDef	→ FN ID OPAREN parameterList CPAREN fnReturn OBRACE Statements CBRACE
fnReturn	→ -> Type | ε
Statements	→ Stmt moreStmts
moreStmts	→ Stmt moreStmts | ε
Stmt	→ COMMENT | ID IDStmts | IfStmt | IStmt | BreakStmt | ReturnStmt | LoopStmt | OStmt | DeclarationStmt | AssignStmtType2
ReturnStmt	→ RETURN relType SEMICOLON
BreakStmt	→ Break SEMICOLON
DeclarationStmt	→ LET mutMod moreDeclarations SEMICOLON
moreDeclarations	→ Declaration | AssignStmtType2
mutMod	→ MUT | ε
Declaration	→ ID ASSIGNOP arithExpn moreTypes
moreTypes	→ COMMA Declaration | ε

```
AssignStmtType2  → listTypes ASSIGNOP OPAREN multAssn CPAREN SEMICOLON
listTypes        → OPAREN typeList CPAREN
typeList         → ID moreList
moreList         → COMMA typeList | ε
singleAssn       → arithExpn
multAssn         → singleAssn moreAssn
moreAssn         → COLON multAssn | COMMA multAssn | ε
IDStmts          → OSQUARE ID CSQUARE OSQUARE ID CSQUARE AssignStmtType1
                  | FunCallStmt | AssignStmtType1 | MethodStmt
IDStmts2         → FunCall | MethodCall | Index | ε
Index            → OSQUARE relType CSQUARE moreIndex | ID | NUM
moreIndex        → OSQUARE relType CSQUARE | ε
AssignStmtType1  → ASSIGNOP singleAssn SEMICOLON
FunCall          → OPAREN IDList CPAREN
MethodCall       → DOT ID FunCall
FunCallStmt      → FunCall SEMICOLON
MethodStmt       → MethodCall SEMICOLON
Type             → BOOL | F32 | I32 | string
parameterList    → ID COLON type remainingList
remainingList    → COMMA parameterList | ε
IfStmt           → IF OPAREN boolExpn CPAREN OBRACE Statements CBRACE
                   ElseStmt
ElseStmt         → ELSEIF OPAREN boolExpn CPAREN OBRACE Statements
                   CBRACE ElseStmt | ELSE OBRACE Statements CBRACE | ε
IStmt            → SCAN OPAREN ID CPAREN SEMICOLON
OStmt            → PRINT OPAREN ID CPAREN SEMICOLON
value            → CHARL | NUM | STRINGL | boolean | array | grid |
                   FLOAT
array            → OSQUARE multAssn CSQUARE
IDList           → ID moreIds | ε
moreIds          → COMMA IDList | ε
arithExpn        → arithTerm moreTerms
moreTerms        → opLow arithExpn | ε
arithTerm        → factor moreFactors
moreFactors      → opHigh arithTerm | ε
factor           → OPAREN arithExpn CPAREN | relType
opLow            → MINUS | PLUS
relType          → ID IDStmts2 | value
opHigh           → DIV | MUL
boolExpn         → OPAREN boolExpn CPAREN logicalOp OPAREN boolExpn
```

```
                    CPAREN | relType relationalOp relType
   logicalOp        → AND | NOT | OR
   relationalOp     → EQ | GT | GTE | LT | LTE | NE
   LoopStmt         → WHILE OPAREN boolExpn CPAREN OBRACE Statements CBRACE
   grid             → OBRACE rows CBRACE
   rows             → row moreRows
   moreRows         → COLON rows | ε
   row              → NUM moreNums
   moreNums         → COMMA NUM moreNums | ε
   boolean          → FALSE | TRUE
}
```

Test Cases

```
// #1  Sum upto n numbers
// Features - loop, I/O, multiple-assignment

fn main() {
     let mut (i, n, sum) = (1, 0, 0);
     scan(n);
     while(i < n) {
         sum = sum +  i;
          i = i + 1;
     }
     print(sum);
}
```

```
// #2 Check if string is Palindrome
// Features - loop, string, boolean

fn main() {
     let mut str1 = "radar";
     if(str1 == str1.reverse()){
         print("Yes");
      }
}
```

```
// #3 Add using functions
// Features - loop, I/O, if, functions

fn add(x:i32, y:i32) -> i32 {
    let mut ans = 0;
    ans = x + y;
    return ans;
}

fn main (){
    let mut x = 0;
    x = add(1, 2);
    print(x);
}
```

```
// #4 Find minimum element in array
// Features - loop, arrays

fn main() {
     let arr = [1, 1, 2, 5];
      let mut i = 1;
     let mut min = arr[0];
     while(i < arr.size()) {
          if(arr[i] < min) {
            min = arr[i];
          }
          i++;
     }

}
```

```
// #5 Print String in uppercase
// Feature String

fn  main(){
    let mut s = "Hello";
    s.toUpper();
    print(s);
}
```