Team Members:

Hajar El Boutahiri

Youssef Yousfi

Hiba Aqqaoui

Part 2 - Language Design

Professor:

Violetta Cavalli-Sforza

CSC 3315 02

Part 1: Lexemes:

<table>
<tr><th>Token (Category)</th><th>Regular Expression (formal specification)</th><th>Example Lexeme(s)</th><th>Remarks</th></tr>
<tr><td>Reserved word

BACK_RES
MAIN_RES
SET_RES
WHILE_RES
END_RES
VOID_RES
INT_RES
BOOL_RES
CHAR_RES
BEGIN_RES
IF_RES
ELSE_RES
CONST_RES
GOUP_RES
GOD_RES
GOL_RES
GOR_RES
READ_RES
PRINT_RES
PICKF_RES
BOOL_LIT</td><td>back|main| if [.]$^{+}$ (else[.]$^{+}$)$^{+}$|
WHILE|main|SET
|END |BEGIN|void|
int|bool | const char
/* | char</td><td>if (a>b) [
b++;
]
Else[
a++;
]

int main (void){
back 0;
}</td><td>back => (Alternative to return in C)
if... else... => for the if-else statement
SET=> to initialize a statement to a value
WHILE => For the loop
END=> Mark the end of the while loop
main=> for the main function
void => for functions that returns nothing and that takes no argument
int => for positive integers
bool => for Boolean values 2 and 3
char => to declare one character
ctchar => to declare string constants
GOUP
GODOWN
GOLEFT
GORIGHT
print
read
=>go in assigned direction
PICKFLOWER</td></tr>
<tr><td>Operators
MULT_OP
ADD_OP
SUB_OP
DIV_OP
EQL_OP
NEQL_OP
ASSIGN_OP
BTOE_OP</td><td>/. | +| -| / |= {1,2}
|<| >|<=|>=</td><td>
1 + 4
5 . 4
b = 2
10 % 2</td><td>. => for multiplication
+ => for addition
- => for subtraction
/ => for division
= => equality operator
=! =>inequality operator</td></tr>
</table>

<table>
<tr><td>LTOE_OP
BT_OP
LT_OP</td><td></td><td></td><td></td></tr>
<tr><td>White spaces</td><td>[ ]*</td><td>SET a = b ;</td><td>“ ”</td></tr>
<tr><td>Punctuation

L_PAREN
R_PAREN
LCBRK
RCBRK
LSQR
RSQR
S_COLON
Q_MARKS
COLON</td><td>,
| (|)
| { | }
|[|]
|;</td><td>int r , j ;
int main (void) {
}
if (a<b) [
]</td><td>() for conditions
{} for statements
; => mark end of statement
, => Separator
[] => limit conditional statements</td></tr>
<tr><td>Positive Decimal Integer Literal
INT_LIT</td><td>0| [1-9][0-9]*</td><td>SET flower = 15;
SET grass = 6;
SET flower_beds = 9;</td><td>003 or 02 etc. Are not allowed. In addition to negative integers</td></tr>
<tr><td>boolean literal
BOOL_LIT</td><td>True | False</td><td>bool a= True;</td><td>True
False</td></tr>
<tr><td>Strings constants</td><td>" [a-zA-Z0-9]* "</td><td>“Hello Flower World”</td><td>zero or more character within double quotes</td></tr>
<tr><td>ID</td><td>[a-zA-Z][_a-zA-Z0-9]{0, 15}</td><td>Count
Id_1
incrementor</td><td>+ An Identifier can’t be a keyword
+Identifiers are case-sensitive
+ An identifier can be alphanumeric; however, it must start with a letter or underscore</td></tr>
</table>

			+Identifier does not include Whitespaces or special characters as: !, @, #, $...

Part 2: Syntax:

// The code contains global variable declarations / function prototypes, main function, function definitions

Code:: = { < type_def > S_COLON} <main> {< type_def > <arg> <function>) }

//Main Definition

<main>::= INT_RES MAIN_RES L_PAREN VOID _RES R_PAREN LCBRK <statement> {<statement>} RCBRK

// Global Variable declaration

<type_def>::= (CONST_CHAR | BOOL_RES | INT_RES) ID

// function definition

<arg>::= L_PAREN (<type_def > { COLON < type_def >}) | VOID_RES R_PAREN

<void> ::= VOID _RES ID <arg>

<function>::= LCBRK <statement> {<statement>} RCBRK

// The different statements that a function may contain

<statement>::= <initialize> | <assign> | <conditional statement> | <loop> | <function_call> | <back>| <predef_func>

//declare a variable inside a function

<initialize>::= <type_def> S_COLON

// assign a value to a variable

<assign>::= SET_RES ID <EQL_OP> ((ID | | INT_LIT) [<operator> (ID | | INT_LIT)]) | (Q_MARKS (<sentence>) Q_MARKS) | (<function_call>)) S_COLON

<operator>::= ADD_OP| SUB_OP| DIV_OP|MULT_OP|MOD_OP

<sentence>::= ID { } { ID { } }

// if.... else... statements

<conditionalstatement>::= IF L_PAREN <condition> R_PAREN LSQR THEN <statement> {<statement>} RSQR [ELSE_RES LSQR <statement> {<statement>} RSQR]

//Put a condition; as an example b > c

<condition>::= (ID| INT_LIT) <conditionaloperator> (ID| INT_LIT)

<conditionaloperator>::= BT_OP | LTOE_OP| BTOE_OP| LT_OP | EQL_OP | NEQL_OP

//While loop

<loop>::= WHILE_RES L_PAREN <condition> R_PAREN BEGIN_RES <statement> {<statement>} END_RES S_COLON

// call a function inside our program

<function_call>::= ID L_PAREN [(ID| INT_LIT){COLON (ID| INT_LIT) }] R_PAREN S_COLON

// call predefined functions inside our program

<predef_func>::= ((PRINT_RES | READ_RES) L_PAREN ID R_PAREN S_COLON) | (GOU_RES| GOD_RES| GOL_RES| GOR_RES| PICKF_RES)L_PAREN R_PAREN S_COLON

// Back statement similar to return statement in C

<back>::= BACK_RES [<back_statement>] S_COLON

back_statement>::= ID | INT_LIT

Part 3:

- **A full set of lexemes for closed categories and representative examples categories like user-defined identifiers and constants. These will be formally specified as regular expressions that will be used by the lexer (as shown above).**

* lexemes:

User defined identifiers: flower, flowerbed, wall, pathway.

User defined constants: WIDTH, HEIGHT, flowerbedloc, flowerloc, grassloc

Predefined functions: PICKFLOWER( ), GOUP( ), GODOWN( ), GOLEFT( ), GORIGHT( ), read(), print()...

Predefined Words: int, main, void, back, char, bool, ctchar , WHILE, END, SET, if, else.

User defined functions: FINDFLOWER(), IFWALLTURN()...

Part 4:

- **Examples for every grammar rule you used to describe the structure of your program. The parts of the example that are described by rules for non-terminals in the RHSs can be expanded with the examples you give for those non-terminals.**

Code:	**GRAMMAR RULES USED:**
int glob;	Code:: = { < type_def > S_COLON} <main> {< type_def > <arg> <function>) } <type_def>::= (INT_RES) ID
int main (void) {	=> In this part of the code, we used the following grammar for the main Function: Code:: = { < type_def > S_COLON} <main> {< type_def > <arg> <function>) } <main>::= INT_RES MAIN_RES L_PAREN VOID _RES R_PAREN LCBRK <statement> {<statement>} RCBRK

bool a;	=> In this part of the code, we used the following parts of the EBNF: <statement>::= <initialize> <initialize>::= <type_def> S_COLON <type_def>:: = BOOL_RES ID
SET ab= pick (5, 10);	=> In this part of the code, we used the following parts of the EBNF: <statement>::= <assign> <assign>::= SET_RES ID <EQL_OP> <function_call>S_COLON <function_call>::= ID L_PAREN [INT_LIT{COLON (INT_LIT) }] R_PAREN S_COLON
back 0;	For this statement we used this part of the EBNF: <statement>::= <back> <back>::= BACK_RES[<back_statement>] S_COLON <back_statement>::= INT_LIT
}	=> In this part of the code, to end the main function we used the following part of the EBNF: <main>::= INT_RES MAIN_RES L_PAREN VOID _RES R_PAREN LCBRK <statement> {<statement>} RCBRK

<table>
<tr><td></td><td></td></tr>
<tr><td>bool PICKFLOWER (int flowerposition, int dimension) {</td><td>For this part of the function definition, we used the following parts of the EBNF:
Code:: = { < type_def > S_COLON} <main> {< type_def > <arg> <function>) }

<arg>::= L_PAREN (<type_def > { COLON < type_def >}) | VOID_RES R_PAREN

<type_def>::= (CONST_CHAR | BOOL_RES | INT_RES) ID
<void> ::= VOID _RES ID <arg>
<function>::= LCBRK <statement> {<statement>} RCBRK</td></tr>
<tr><td>int count;</td><td>For this initialisation we used those parts of the EBNF:
<statement>::= <initialize>
<initialize>::= <type_def> S_COLON
<type_def>:: = INT_RES ID</td></tr>
<tr><td>ctchar hello;
SET hello = “Hello Flower World”;
ctchar found;
SET found = “Flower picked”;</td><td>For this initialisation we used this part of the EBNF:
<statement>::= <initialize>
<initialize>::= <type_def> S_COLON
<type_def>:: = CONST_CHAR ID
<assign>::= SET_RES ID EQL_OP Q_MARKS (<sentence>) Q_MARKS</td></tr>
<tr><td>print (hello);</td><td>In this part, we used the following parts of our EBNF:
<statement>::= <predef_func></td></tr>
</table>

	<predef_func>::= ((PRINT_RES | READ_RES) L_PAREN ID R_PAREN S_COLON)
SET count = 0;	For this statement we used the following parts of the EBNF: <statement>::= <assign> <assign>::= SET_RES ID EQL_OP INT_LIT S_COLON
WHILE (count < dimension)	For this statement we used the following parts of the EBNF: <loop>::= WHILE_RES L_PAREN <condition> R_PAREN BEGIN_RES <statement> {<statement>} END_RES S_COLON <condition>::= ID <conditionaloperator> ID <conditionaloperator>::= LT_OP
BEGIN	For this statement we used the following parts of the EBNF: <loop>::= WHILE_RES L_PAREN <condition> R_PAREN BEGIN_RES <statement> {<statement>} END_RES S_COLON
if (count == flowerposition)[	=> For this condition we used this part of the EBNF: <statement>::= <conditional statement> <conditionalstatement>::= IF L_PAREN <condition> R_PAREN LSQR THEN <statement> {<statement>} RSQR [ELSE_RES LSQR <statement> {<statement>} RSQR] <condition>::= (ID| INT_LIT) <conditionaloperator> (ID| INT_LIT)

	<conditionaloperator>::= EQL_OP
print (found);	In this part, we used the following parts of our EBNF: <statement>::= <predef_func> <predef_func>::= ((PRINT_RES | READ_RES) L_PAREN ID R_PAREN S_COLON)
back 2;	=> For this statement we used this part of the EBNF: <statement>::= <back> <back>::= BACK_RES [<back_statement>] S_COLON <back_statement>::= INT_LIT
]	=>To end an if statement according to our grammar we need a right square bracket: RSQR: => For this condition we used this part of the EBNF: <conditionalstatement>::= IF L_PAREN <condition> R_PAREN LSQR THEN <statement> {<statement>} RSQR [ELSE_RES LSQR <statement> {<statement>} RSQR]
else [	=> For an Else if statement according to our grammar we need a left square bracket: LSQR: <conditionalstatement>::= IF L_PAREN <condition> R_PAREN LSQR THEN <statement> {<statement>} RSQR [ELSE_RES LSQR <statement> {<statement>} RSQR] => For else, we don’t need a conditionnal statement
SET count = count + 1;	=> In this part of the code, we used the following parts of the EBNF: <statement>::= <assign>

	<assign>::= SET_RES ID EQUAL (ID | INT_LIT) { <operator> (ID| INT_LIT)} <operator>::= ADD_OP
]	=>To end an else statement according to our grammar we need a right square bracket: RSQR: <conditionalstatement>::= IF L_PAREN <condition> R_PAREN LSQR THEN <statement> {<statement>} RSQR [ELSE_RES LSQR <statement> {<statement>} RSQR]
END;	=> In our grammar, a loop is marked by END; <loop>::= WHILE_RES L_PAREN <condition> R_PAREN BEGIN_RES <statement> {<statement>} END_RES S_COLON
back 3;	=> For this statement we used this part of the EBNF: <statement>::= <back> <back>::= BACK_RES [<back_statement>] S_COLON <back_statement>::= INT_LIT
}	=> To end a function definition in our grammar we need a RCBRK: <function>::= LCBRK <statement> {<statement>} RCBRK

Part 5:

- **Examples of simpler and more complex programs. For example:**

a. A program that moves the agent a few steps only:

```
int main(void){
  int positionX;
```

```
    int positionY;
    SET positionX = 3;
    SET positionY = 7;


    SET i = 0;
    WHILE (i<2)
    BEGIN
            GORIGHT();
             SET i = i + 1;
    END;

    SET i = 0;
    WHILE (i<3)
    BEGIN
           GOUP();
            SET i = i + 1;
    END;

    back 0;
}
```

b. A program that has an agent pick all flowers from a flowerbed:

```
int main(void){

    int positionX;
    int positionY;
    int LLx;
    int LLy;
    int ULx;
    int ULy;
    int URx;
    int URy;
    int LRx;
    int LRy;

    SET LLx = 8;
    SET LLy = 1;
    SET ULx = 8;
    SET ULy = 3;
    SET URx = 12;
    SET URy = 3;
    SET LRx = 12;
    SET LRy = 1;

    SET positionX = 8;
```

```
SET positionY = 0;


SET i = 0;
WHILE (i < LRx)
BEGIN
       PICKFLOWER();
       GORIGHT();
       SET i = i + 1;
END;

GORIGHT();
GOUP();
GOUP();

SET i = 0;
WHILE (i < URy)
BEGIN
       PICKFLOWER();
       GOUP();
       SET i = i + 1;
END;

GOUP();
GOLEFT();
GOLEFT();

SET i = 0;
WHILE (i < LRx)
BEGIN
      PICKFLOWER();
      GOLEFT();
      SET i = i + 1;
END;

GOLEFT();
GODOWN();
GODOWN();

SET i = 0;
WHILE (i < LRx)
BEGIN
       PICKFLOWER();
       GODOWN();
         SET i = i + 1;
```

```
    END;

    back 0;
}
```

c. A program that navigates the whole Flower World, picking all the flowers and exits:

```
  int positionX;
  int positionY;
  int flowerCount;
int main(void){
    SET positionX = 0;
    SET positionY = 0;
    SET flowerCount = 0;

    ctchar Exit;
    SET Exit = "Left The Flower World with ";

    ctchar Flowers;
    SET FlowerCount = " Flowers";


    SET flowerCount = traverse(positionX, positionY);

    print(Exit );
   print(flowerCount);

    back 0;
}
```

d. a program that cannot exit because the path to the exit is blocked by flowerbeds:

```
int main(void){

    int positionX;
    int positionY;

    SET positionX = 0;
    SET positionY = 0;

    bool foundExit;

    ctchar noExit;
    SET noExit = "Exit Blocked by Flowerbeds";
```

```
    SET foundExit = traverse(positionX, positionY);

    if (foundExit == 3) [
            print(noExit);
]

    back 0;
}
```