

## **Team Members:**

Hajar El Boutahiri

Youssef Yousfi

Hiba Aqqaoui

Part 2 - Language Design

### **Professor:**

Violetta Cavalli-Sforza

CSC 3315 02

## Part 1: Lexemes:

Token	Regular	Example	Remarks
(Category)	Expression	Lexeme(s)	
	(formal		
	specification)		
Reserved word	back main  if [.] <sup>+</sup>	<b>if</b> (a>b) [	back => (Alternative to
	(else[.] <sup>+</sup> ) <sup>+</sup>	b++;	return in C)
	WHILE main SET		if else => for the if-else
	END  BEGIN void    int bool   const char	Else[ a++;	statement SET=> to initialize a
	/*   char	1	statement to a value
		'	WHILE => For the loop
BACK_RES			END=> Mark the end of the
MAIN_RES			while loop
SET_RES		int main (void){ back 0;	main=> for the main function
WHILE_RES		Back U,	void => for functions that
END_RES VOID_RES		,	returns nothing and that
INT RES			takes no argument
BOOL_RES			int => for positive integers
CHAR_RES			bool => for Boolean values 2 and 3
BEGIN_RES			char => to declare one
IF_RES			character
ELSE_RES			<pre>ctchar =&gt; to declare</pre>
CONST_RES GOUP_RES			string constants
GOD_RES			GOUP GODOWN
GOL_RES			GOLEFT
GOR_RES			GORIGHT
READ_RES			print
PRINT_RES			read
PICKF_RES			=>go in assigned direction
BOOL_LIT			PICKFLOWER
Operators	/.   +  -  /  = {1,2}		. => for multiplication
MULT_OP	< > <= >=	1 + 4	+ => for addition
ADD_OP		5 . 4 b = 2	- => for subtraction / => for division
SUB_OP DIV_OP		10 % 2	= => equality operator
EQL_OP			=! =>inequality operator
NEQL_OP			, , ,
ASSIGN_OP			
BTOE_OP			

LTOE_OP BT_OP LT_OP			T	
TT_OP   White spaces   []*   SET a = b; " "				
White spaces  []*  SET a = b; ""  Punctuation    ( )				
Punctuation  Punctuation    (   )	LT_OP			
Punctuation    ( )	White spaces	[]*	SFT a = b:	u n
L_PAREN	vvinte spaces		02. 0	
L_PAREN R_PAREN   {   }	Punctuation	,	int r , j ;	() for conditions
R_PAREN LCBRK RCBRK LSQR RSQR S_COLON Q_MARKS COLON  Positive Decimal Integer Literal INT_LIT    Decimal literal BOOL_LIT    Decimal literal BOOL_LIT   Decimal literal BOOL_LIT   Decimal literal BOOL_LIT   Decimal literal False    Decimal literal   Decimal l		(   )	int main (void) {	{} for statements
CCBRK   CBRK   LSQR   RSQR   S_COLON   COLON	L_PAREN	{   }	}	; => mark end of statement
RCBRK LSQR RSQR S_COLON Q_MARKS COLON  Positive Decimal Integer Literal INT_LIT  Positive Decimal Integer Literal INT_LIT  SET flower = 15; SET flower_beds = 9;  boolean literal BOOL_LIT  BOOL_LIT  True   False  bool a= True; False  True   False  Strings constants  " [a-zA-Z0-9]* " "Hello Flower World"  The positive integers  True   False  True   False  True   False  Count Id_1 incrementor    An Identifier can't be a keyword   Heldentifiers are casesensitive	R_PAREN		if (a <b) [<="" td=""><td>•</td></b)>	•
LSQR RSQR S_COLON Q_MARKS COLON  Positive Decimal Integer Literal INT_LIT  Positive Decimal Integer Literal INT_LIT    0   [1-9][0-9]*   SET flower = 15; SET grass = 6; SET flower_beds = 9;   SET flower_beds = 9;	LCBRK	;	]	
RSQR S_COLON Q_MARKS COLON  Positive Decimal Integer Literal INT_LIT  Positive Decimal Integer Literal INT_LIT  SET grass = 6; SET flower_beds = 9;  boolean literal BOOL_LIT  True   False  bool a= True; False  Strings constants  " [a-zA-Z0-9]*"  "Hello Flower World"  "Hello Flower within double quotes  True   False  "Hello Flower within double quotes  "An Identifier can't be a keyword + Identifiers are case-sensitive + An identifier can be alphanumeric; however, it must start with a letter or	RCBRK			statements
S_COLON Q_MARKS COLON  Positive Decimal Integer Literal INT_LIT    O   [1-9][0-9]*   SET flower = 15; SET grass = 6; SET flower_beds = 9;   Dool and Iteral BOOL_LIT   Dool and Iteral BOOL_LIT   True   False   False   False   True; False   True   True   False   True   True   False   True   False   True   Tr	=			
Positive Decimal Integer Literal INT_LIT  Positive Decimal Integer Literal INT_LIT  O  [1-9][0-9]*  SET flower = 15; SET grass = 6; SET flower_beds = 9;  boolean literal BOOL_LIT  True   False  bool a= True; True False  "[a-zA-Z0-9]*"  "Hello Flower World"  zero or more character within double quotes  ID  [a-zA-Z][_a-zA-Z0-9]*    An Identifier can't be a keyword + Identifiers are case-sensitive + An identifier can be alphanumeric; however, it must start with a letter or	~			
Positive Decimal Integer Literal INT_LIT  Positive Decimal Integer Literal INT_LIT  SET grass = 6; SET flower_beds = 9;  boolean literal BOOL_LIT  True   False  bool a= True;  "[a-zA-Z0-9]*"  "Hello Flower World"  True   False  "[a-zA-Z][_a-zA-Z0-9]*"  ID  [a-zA-Z][_a-zA-Z0-9]*  [a-zA-Z][_a-zA-Z0-9]*  [a-zA-z][_a-zA-zo-9]*  Han Identifier can't be a keyword + Identifiers are case-sensitive + An identifier can be alphanumeric; however, it must start with a letter or				
Positive Decimal Integer Literal INT_LIT    SET grass = 6; SET flower_beds = 9;   SET flower_beds = 9;   O03 or 02 etc. Are not allowed. In addition to negative integers	-			
INT_LIT  SET grass = 6; SET flower_beds = 9;  boolean literal BOOL_LIT  True   False  bool a= True; False  "[a-zA-Z0-9]*"  "Hello Flower World"  "[a-zA-Z1][a-zA-Z0-9]*"  Ld_1 incrementor  False  True   False  True   False  "An Identifier can't be a keyword + Identifiers are casesensitive + An identifier can be alphanumeric; however, it must start with a letter or	COLON			
INT_LIT  SET grass = 6; SET flower_beds = 9;  boolean literal BOOL_LIT  True   False  bool a= True; False  "[a-zA-Z0-9]*"  "Hello Flower World"  "[a-zA-Z1][a-zA-Z0-9]*"  Ld_1 incrementor  False  True   False  True   False  "An Identifier can't be a keyword + Identifiers are casesensitive + An identifier can be alphanumeric; however, it must start with a letter or	Positive Decimal Integer Literal	01 [1-0][0-0]*	SET flower - 15:	003 or 02 etc. Are not
SET flower_beds   = 9;			-	
boolean literal BOOL_LIT  True   False  bool a= True; False  Strings constants  " [a-zA-Z0-9]* "  [a-zA-Z][_a-zA-Z0-9]* "  Count   d_1   incrementor    ld_normementor   ld_normementor   ld_normementor	IIVI_EII			
boolean literal BOOL_LIT  True   False  bool a= True; False  Strings constants  " [a-zA-Z0-9]* "  "Hello Flower World"  "Count Id_1 incrementor    Id_1 incrementor			_	
Strings constants  " [a-zA-Z0-9]* "  "Hello Flower World"  zero or more character within double quotes  ID  [a-zA-Z][_a-zA-Z0-9]*   Count Id_1 incrementor   + An Identifier can't be a keyword   + Identifiers are casesensitive   + An identifier can be alphanumeric; however, it must start with a letter or			- 3,	
Strings constants  " [a-zA-Z0-9]* "  "Hello Flower World"  zero or more character within double quotes  ID  [a-zA-Z][_a-zA-Z0-9]*   Count Id_1 incrementor   + An Identifier can't be a keyword   + Identifiers are casesensitive   + An identifier can be alphanumeric; however, it must start with a letter or	boolean literal	True   False	bool a= True;	True
Strings constants  " [a-zA-Z0-9]* "  "Hello Flower World"  Zero or more character within double quotes  TD  [a-zA-Z][_a-zA-Z0-9]{0, 15}  Count Id_1 hand ldentifier can't be a keyword heldentifiers are casesensitive han identifier can be alphanumeric; however, it must start with a letter or		·	·	False
World" within double quotes  ID	_			
World" within double quotes  ID				
ID  [a-zA-Z][_a-zA-Z0- 9]{0, 15}  Count Id_1	Strings constants	" [a-zA-Z0-9]* "		
Id_1 incrementor  Id_1 the first are can't be a keyword the first are case- sensitive the first are case- sensitive the first are can't be a keyword the first are case- sensitive the first are can't be a keyword the first are can't be a keyword the first are case- sensitive the first are can't be a keyword the first are case- sensitive the first are can't be a keyword the first are case- sensitive the first are can't be a keyword the first are case- sensitive the first are can't be a keyword the first are case- sensitive			World"	within double quotes
Id_1 incrementor    An Identifier can't be a keyword				
Id_1 incrementor    An Identifier can't be a keyword	10			
Incrementar   keyword   +Identifiers are case-sensitive   + An identifier can be   alphanumeric; however, it   must start with a letter or	ID			+ An Identifier can't be a
+Identifiers are case- sensitive  + An identifier can be alphanumeric; however, it must start with a letter or		9]{0, 15}	_	keyword
sensitive + An identifier can be alphanumeric; however, it must start with a letter or				
+ An identifier can be alphanumeric; however, it must start with a letter or				
alphanumeric; however, it must start with a letter or				sensitive
must start with a letter or				+ An identifier can be
				alphanumeric; however, it
				must start with a letter or
underscore				underscore

	+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> {&lt;</main>
	type_def > <arg> <function> ) }</function></arg>
	<type_def>::= (INT_RES) ID</type_def>
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> ) }</function></arg>
	<main>::= INT_RES MAIN_RES L_PAREN VOID</main>
	_RES R_PAREN LCBRK <statement></statement>
	{ <statement>} RCBRK</statement>

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

bool PICKFLOWER (int flowerposition, int dimension) {	For this part of the function definition, we used the following parts of the EBNF:  Code:: = { < type_def > S_COLON} <main> { &lt; type_def &gt; <arg> <function> ) }  <arg>::= L_PAREN (<type_def> { COLON &lt; type_def &gt;} )   VOID_RES R_PAREN  <type_def>::= (CONST_CHAR   BOOL_RES  </type_def></type_def></arg></function></arg></main>
	<pre>INT_RES) ID  <void> ::= VOID _RES ID <arg>  <function> ::= LCBRK <statement> {<statement>} RCBRK</statement></statement></function></arg></void></pre>
int count;	For this initialisation we used those parts of the EBNF: <statement>::= <initialize> <initialize>::= <type_def> S_COLON  <type_def>::= INT_RES ID</type_def></type_def></initialize></initialize></statement>
ctchar hello;  SET hello = "Hello Flower World";  ctchar found;  SET found = "Flower picked";	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</sentence></assign></type_def></type_def></initialize></initialize></statement>
print (hello);	In this part, we used the following parts of our EBNF: <statement>::= <pre><pre>cstatement&gt;::= <pre>cstatement&gt;::= <pre>cstat</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></statement>

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

	conditional operators FOL OD
	<pre><conditionaloperator>::= EQL_OP</conditionaloperator></pre>
print (found);	In this part, we used the following parts of our EBNF:
	<statement>::= <predef_func></predef_func></statement>
	. 1 C C . (/DDD/E DEG   DEAD DEG)
	<pre><pre><pre><pre><pre><pre><pre>f_func&gt;::= ((PRINT_RES   READ_RES)</pre> <pre>L_PAREN ID R_PAREN S_COLON)</pre></pre></pre></pre></pre></pre></pre>
back 2;	=> For this statement we used this part of the EBNF:
buck 2,	-> 1 of this statement we used this part of the EDIVI.
	<statement>::= <back></back></statement>
	<pre><back>::= BACK_RES [<back_statement>]</back_statement></back></pre>
	S_COLON
	chook statements up INIT LIT
	<back_statement>::= INT_LIT</back_statement>
1	=>To end an if statement according to our grammar
,	7 TO ONG UN IT SUCCESSION GOOD STAIRMAN
	we need a right square bracket: RSQR:
	=> For this condition we used this part of the EBNF:
	THE TAX DARRENT HE
	<pre><conditionalstatement>::= IF L_PAREN <condition></condition></conditionalstatement></pre>
	R_PAREN LSQR THEN <statement> {<statement>}</statement></statement>
	RSQR [ ELSE_RES LSQR <statement></statement>
	{ <statement>} RSQR ]</statement>
else [	=> For an Else if statement according to our grammar
	we need a left square bracket: LSQR:
	<pre><conditionalstatement>::= IF L_PAREN <condition></condition></conditionalstatement></pre>
	R_PAREN LSQR THEN <statement> {<statement>}</statement></statement>
	RSQR [ ELSE_RES LSQR <statement></statement>
	{ <statement>} RSQR ]</statement>
	=> For else, we don't need a conditionnal statement
SET count = count + 1;	=> In this part of the code, we used the following
	ports of the EDNE.
	parts of the EBNF:
	<statement>::= <assign></assign></statement>
	Summing Subsigni

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

# 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 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;

```
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;
```

```
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;
}
```