uc3m | Universidad Carlos III de Madrid

DEPARTMENT OF COMPUTER SCIENCE
CARLOS III UNIVERSITY OF MADRID

Computer Science
Language Processors

Rules

- The duration of the test is **3.0 hours**
- Questions will not be answered during the test
- One cannot re-enter the classroom after leaving it
- The answers must be written using a pen (not a pencil)

Problem 1. Recursive Descent Parsing (3.5 points)

Given the following Grammar:

```
Expression      ::=  ( Operator Operand MoreOperands )
Operator        ::=  +
Operator        ::=  *
Operand         ::=  Number | Expression
MoreOperands    ::=  Number
MoreOperands    ::=  Number MoreOperands
```

You are asked to:

a) **Transform the grammar if necessary.**

b) **Calculate de First and Follow Sets.**

c) **Design a Recursive Descent Parser.**

d) **How can we control the correct application of operator precedence?**

The token and the literals of the language are *Number* , (,) , + and *
You are not required to include semantic routines (the program should not calculate anything, just perform parsing).

Problem 2. Bottom Up Parsing (3 points)

Given the following Grammar:

Expression	**::=**	**(Operator Operand Operand)**
Operator	**::=**	**+**
Operator	**::=**	**-**
Operand	**::=**	**Number**
Operand	**::=**	**Expression**

You are asked to:

a) **Obtain the canonical configuration sets LR (0) and the corresponding automaton.**

b) **Obtain the LR(0) parser tables.**

c) **Obtain the SLR parser tables.**

Problem 3. Grammar Design and Semantic Analysis (3.5 points)

You are asked to design a very Elementary calculator to process complex numbers. The chosen representation for these numbers will be: <p.real> + <p.imaginary> i
For example: 3 + 5i
The calculator should be able to perform the following operations:

- **Print the result when the expression finishes with a line break**
- **Add two complex numbers, with the + operator**
- **Multiply two complex numbers, with the * operator**
- **More elaborate expressions can be constructed combining several operators and operands.**
- **The use of parentheses is also allowed to encompass at least one entire complex number.**
- **The input expressions will not allow the use of the unary - sign.**
- **One or more expressions, always separated by line breaks, can be entered and evaluated.**

The sum of two complex numbers is calculated as: a + bi + c + di = (a + c) + (b + d) i
The parentheses are included to differentiate the real part from the imaginary one.
The product of two complex numbers requires the following operation:
a + bi * c + di = (a + bi) * (c + di) = (ac-bd) + (bc + ad) i
Note that the inner + operator has higher precedence than the addition and multiplication operators, since it serves to define a complex number. At the same time. the addition operator has less precedence than the multiplication operator.
We include several examples of inputs and their outputs:

Input	Output
1+2i + 3+4i	4.0+6.0i
1+2i * 3+4i	-5.0+10.0i
1.1+2.2i	1.1+2.2i
1+0i + 0+1i	1.0+1,0i
1+0i * 0+1i	0.0+1.0i
(1+1i)	1.0+1.0i
1+1i * 1+1i	0.0+2.0i

Input	Output
1+2i + 1+2i	2.0+4.0i
1+2i * 1+2i	-3.0+4.0i
1+2i + 1+2i + 1+2i	3.0+6.0i
1+2i + 1+2i * 1+2i	-2.0+6.0i
1+2i * 1+2i + 1+2i	-2.0+6.0i
(1+2i * 1+2i) + 1+2i	-2.0+6.0i
1+2i * (1+2i + 1+2i)	-6.0+8.0i

You are asked to:

a) **Design the grammar to read complex numbers with the indicated format (always whole the part followed by a +, the imaginary part and i), combined in arithmetic expressions**

b) **Add the necessary semantic routines to perform the calculations.**

c) **Define the necessary precedences so that the results are the same as indicated in the table**

Pay attention to the code provided at the end of this statement.
The correct structuring of the grammar and the proper use of the defined structures will be evaluated.

```
%{                                  /* 1  Declarations of C-bison Section */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct s_complex {
      double real ;
      double imaginary ;
} t_complejo ;

#define YYSTYPE  t_complex             /* stack type of the parser         */
%}
                                    /* 2  Declaraciones de bison Section  */
%token NUMBER
```

```
                    . . .

%%                                  /* 3  Syntactic - Semantic Section  */

                    . . .
```

```
%%                                   /* 4  C Code Section */

int yylex ()
{
    int c ;
    double myNUMBER ;

    do {                                                 /* read un char */
         c = getchar () ;
         if (c == EOF)
             return 0 ;

    } while (c == ' ' || c == '\t') ;

    if (c >= '0' && c <= '9') {                      /* it is a number. . .  */
         ungetc (c, stdin) ;
         scanf ("%lf", &myNUMBER) ;
         yylval.real = myNUMBER ;
         return NUMBER ;
    }

    return c ;                   /* o or another literal of type char */
}

int yyerror (char *message)
{
    fprintf (stderr, "Syntactic Error\n") ;
}

int main (int argc, char *argv [])
{
    fprintf (stderr, "Starting parser\n") ;
    yyparse () ;
}
```