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:

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

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

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

h) **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**

CC BY NC SA

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

<u>You are asked to:</u>

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

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

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

<u>Problem 3</u>. 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

<u>**You are asked to:**</u>

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

d) 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

e) Add the necessary semantic routines to perform the calculations.

f) 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.

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber
CC BY NC SA

```
%{                                /* 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 () ;
}
```

CC BY NC SA

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

SOLUTIONS:

Problem 1. Recursive Descent Parsing

a) The Grammar, as formulated, does not serve for a recursive descendent parser. All productions of MoreOperands begin by generating the token NUMBER, so we have to factorize.

```
Expression       ::=   ( Operator Operand MoreOperands )
Operator         ::=   + | *
Operand          ::=   NUMBER | Expression
MoreOperands     ::=   NUMBER RestMOperands
RestMOperands    ::=   λ | MoreOperands
```

b) To implement the parser, we will need the First and Following Sets of the Non-Terminals the Follow Set for those that are nullable (RestMOperands), and the First Set for all the Non-Terminals that derive alternative productions. There are no overlaps between the First and Follow Sets of each Non-Terminal.

		First	Follow
E	**Expression**	(	NUMBER $
O	**Operator**	+ *	(NUMBER
P	**Operand**	NUMBER (	NUMBER
M	**MoreOperands**	NUMBER	)
R	**RestMOperands**	λ NUMBER	)

c) For each Non-Terminal a Parse function is designed. For each of its right parts, the sequence of symbols will result in a sequence of calls to their corresponding Parse functions. Nullable Non-Terminals (R) and those that generate several right parts (O and P) must be treated specifically.

E ::= (O P M)
No alternatives.

```
ParseE () {
      ParseLParen () ;
      ParseOperator () ;
      ParseOperand () ;
      ParseMoreOperands () ;
      ParseRParen () ;
}
```

P ::= NUMBER | Expression
Alternative Productions
⇨ First(P) = { (, NUMBER }

```
ParseP () {
      if (token == T_Number)
            ParseNumber () ;
      else
            ParseExpression () ;
}
```

O ::= + | * Alternative Productions
⇨ First(O) = { +, * }

```
ParseOperator () {
      if (token == T_ADD)
            ParseAdd () ;
      else if (token == T_MULT) {
            ParseMult () ;
      }
}
```

M ::= NUMBER RestMOperands
No alternatives.
R is nullable:
⇨ Following (R) = {) }

```
ParseMoreOperands () {
      ParseNUMBER () ;
      if (token ∉ { ) })
            ParseRestMOperands() ;
}
```

R ::= λ | MoreOperands
Alternative Productions
⇨ Following(R) = { λ, NUMBER}
But λ already treated in the call

```
ParseRestMOperands () {
      ParseMoreOperands () ;
```

The Functions ParseX, where X is a token or literal { ( ) + * NUMBER } call Function MatchToken(X) which also reads the next token.

```
MatchToken (X) {
      if (token != X) error () ;
```

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

cc BY NC SA

`}`	`else token = rd_token () ;` `}`

d) The grammar describes a Language that uses prefix notation. To include the Operands of each operation, it is necessary to use parentheses. The use of parentheses is an alternative to determine the order in which Operators should be applied. Therefore, in this case, ambiguities cannot be given nor any precedence is needed.

Problem 2. Bottom Up Parsing

a)

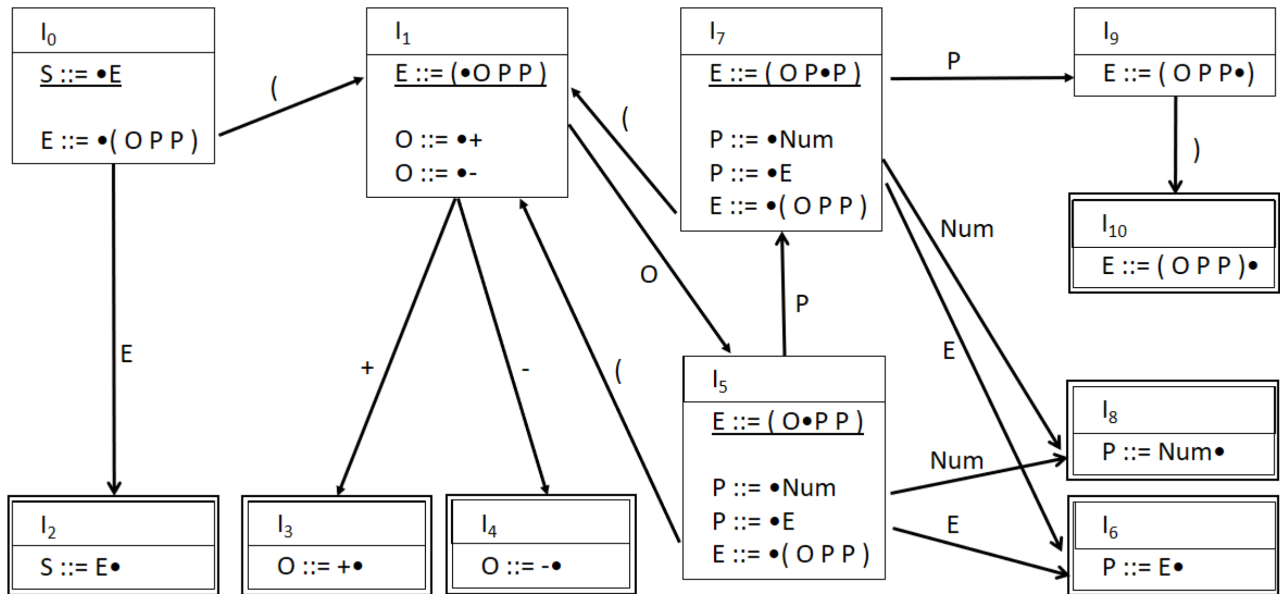
The first step is to augment the grammar with a new Axiom (S):

```
0. S ::= E
1. E ::= ( O P P )
2. O ::= +
3. O ::= -
4. P ::= Num
5. P ::= E
```

To construct the set of configurations we include the augmented production S :: = • E in the initial configuration I_0. We will only add the equivalent items (and not all productions) to obtain:

I_0
S ::= •E
E ::= •(O P P)

Represented as an automata:



b) The LR(0) parse table is:

	(	)	+	-	n	$	E	O	P
0	S1						2		
1			S3	S4				5	
2						acc			
3	R2	R2	R2	R2	R2	R2			
4	R3	R3	R3	R3	R3	R3			
5	S1				S8		6		7
6	R5	R5	R5	R5	R5	R5			
7	S1				S8		6		9
8	R4	R4	R4	R4	R4	R4			
9		S10							
10	R1	R1	R1	R1	R1	R1			

c) The SLR parse table is:

	(	)	+	-	n	$	E	O	P
0	S1						2		
1			S3	S4				5	
2						Acc			
3	R2				R2				
4	R3				R3				
5	S1				S8		6		7
6	R5	R5			R5				
7	S1				S8		6		9
8	R4	R4			R4				
9		S10							
10	R1	R1			R1	R1			

To determine the Reductions in SLR, we rely on the token determined by Following of each reduced Non-

⇨ i.e.:

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

Terminal.

	First	Following
E	(	( ) num $
O	+ -	(num
P	(num	( ) num

R1	`E ::= ( O P P )`	( ) num $
R2	`O ::= +`	(num
R3	`O ::= -`	(num
R4	`P ::= num`	( ) num
R5	`P ::= E`	( ) num

Problem 3. Grammar Design and Semantic Analysis

a) & b)

```
                                    /* 3  Syntactic - Semantic Section   */

axiom:      eval '\n'                { printf("%lf+%lfi\n", $1.real,
$1.imaginary); }
                r_axiom              { ; }
          ;

r_axiom:    // lambda                { ; }
          | axiom                    { ; }
          ;

eval:       complex                  { $$ = $1 ; }
           | '(' eval ')'   '        { $$ = $2 ; }
           | eval '+' eval           {
                                       $$.real = $1.real + $3.real ;
                                       $$.imaginary = $1.imaginary + $3.imaginary
;
                                     }
          | eval '*' eval            {
                                       $$.real = $1.real * $3.real - $1.imaginary
                                       * $3.imaginary ;
                                       $$.imaginary = $3.real * $1.imaginary +
                                       $1.real * $3.imaginary ;
                                     }
          ;

complex:    NUMBER '+' NUMBER 'i'    {
                                       $$.real = $1.real ;
                                       $$.imaginary = $3.real ;
                                     }
          ;
```

There is no need to make any changes to the provided code. The lexical analyzer returns the complex structure always initialized with a real value in the field of the same name. When followed by an 'i' literal, this value must be copied to the imaginary field.

c) To state the correct precedence for the Operators, the following directives are included in the header

```
%left '+'
%left '*'
```

This defines a greater precedence of the Operator * over the Operator +. This will only work if there are ambiguities, and therefore the grammar must have double recursion:

CC BY NC SA

```
eval:       eval '+' eval
          | eval '*' eval
```

The greater precedence of the inner + Operator of a complex number (<real> + <imaginary> i) over the other Operators, is achieved by decomposing the grammar into two hierarchical sub-grammars:

```
eval:       complex
          | eval '+' eval
. . .
complex:    NUMBER '+' NUMBER 'i'
```

We can also define precedence with a contextual token (as in the case of % prec UNARYMINUS).

Trying to use in two different token is not that viable: it would be necessary to modify the lexical analyzer, which in turn would have some difficulties in determining when the Operator + is of inside type and when outside.

cc BY NC SA