MARMARA UNIVERSITY

FACULTY OF ENGINEERING

Project 1_2

Department: CENG
Due Data: 23/05/2023
Used Language: Java

	Dept	Student ID	Name Surname
1	CENG	150121004	Ahmet Arda Nalbant
2	CENG	150121036	Hasan Özeren
3	CENG	150121991	Niyazi Ozan Ateş

PURPOSE

In this project we will implement a syntax analyzer (parser) for a specific programming language. We also call this a recursive-descent parser. We use the tokens that we have found in the first part of the project. The grammar for this Syntax Analyzer is as give below:

<Program> → ε | <TopLevelForm> <Program>

<TopLevelForm> → **(** <SecondLevelForm> **)**

<SecondLevelForm> → <Definition> | **(** <FunCall> **)**

<Definition> → **DEFINE** <DefinitionRight>

<DefinitionRight> → **IDENTIFIER** <Expression> | **(** **IDENTIFIER** <ArgList> **)** <Statements>

<ArgList> → ε | **IDENTIFIER** <ArgList>

<Statements> → <Expression> | <Definition> <Statements>

<Expressions> → ε | <Expression> <Expressions>

<Expression> → **IDENTIFIER** | **NUMBER** | **CHAR** | **BOOLEAN** | **STRING** | **(** <Expr> **)**

<Expr> → <LetExpression> | <CondExpression> | <IfExpression> | <BeginExpression> | <FunCall>

<FunCall> → **IDENTIFIER** <Expressions>

<LetExpression> → **LET** <LetExpr>

<LetExpr> → **(** <VarDefs> **)** <Statements>

| **IDENTIFIER (** <VarDefs> **)** <Statements>

<VarDefs> → **(** **IDENTIFIER** <Expression> **)** <VarDef>

<VarDef> → ε | <VarDefs>

<CondExpression> → **COND** <CondBranches>

<CondBranches> → **(** <Expression> <Statements> **)** <CondBranches>

<CondBranch> → ε | **(** <Expression> <Statements> **)**

<IfExpression> → **IF** <Expression> <Expression> <EndExpression>

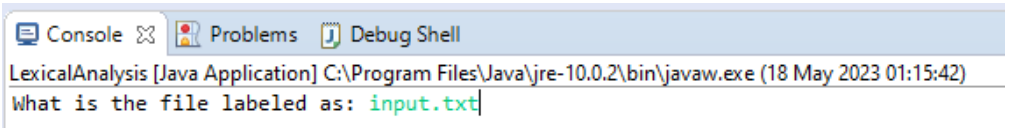
<EndExpression> → ε | <Expression>

<BeginExpression> → **BEGIN** <Statements>

USAGE OF THE CODE

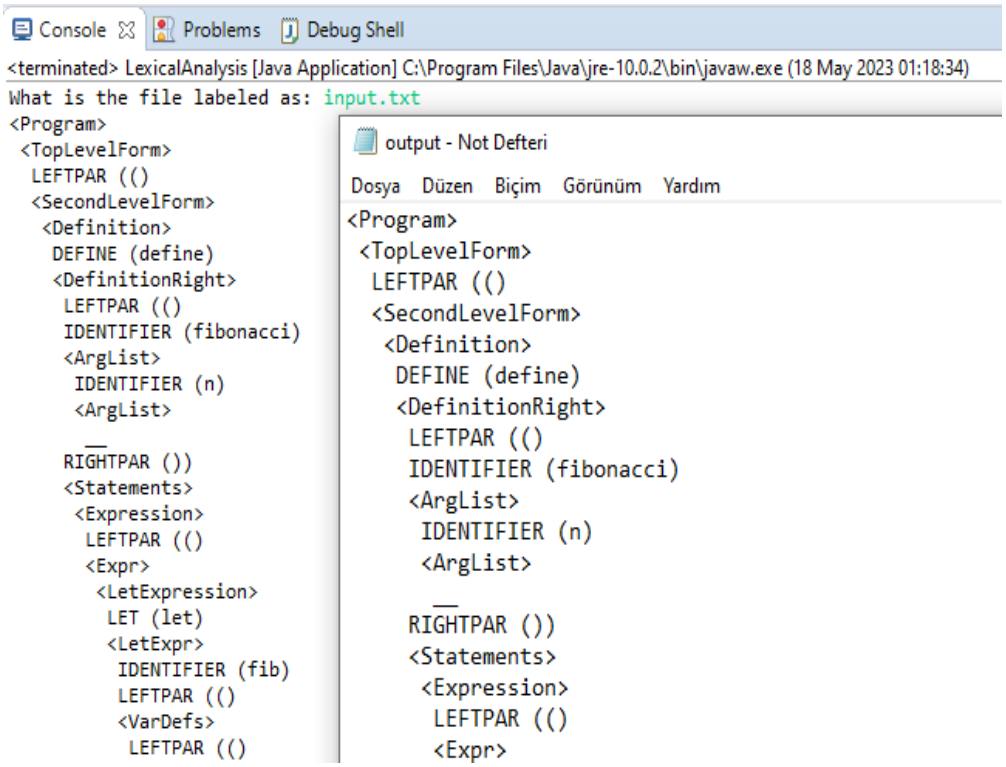
We print our results to an output file based on the given input file. We ask the input file to the user. If this input is lexically incorrect. Then we print out the output as in the first part of this project. Otherwise, we print out the results based on the correct syntax. If it is the case that there is a syntax error, we will report this when it is found.

When running the program, it will ask you to put an input file name asfollows:

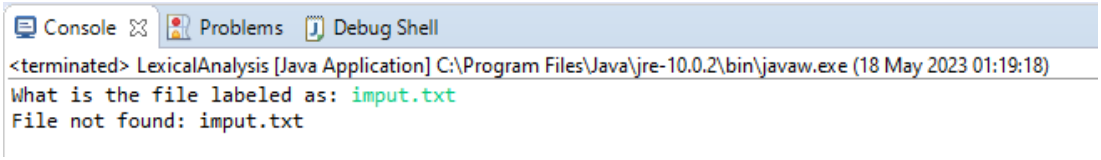


If it is the case that the user enters a valid input text file, then the program will run and gives you the correct output in both console and outputtext file. An example of the input and output is as follows:

```
1 (define (fibonacci n)
2       ( let fib ((prev 0) (cur 1) (i 0))
3             ( if (= i n) cur (fib cur (+ prev cur) (+ i 1)))))
```

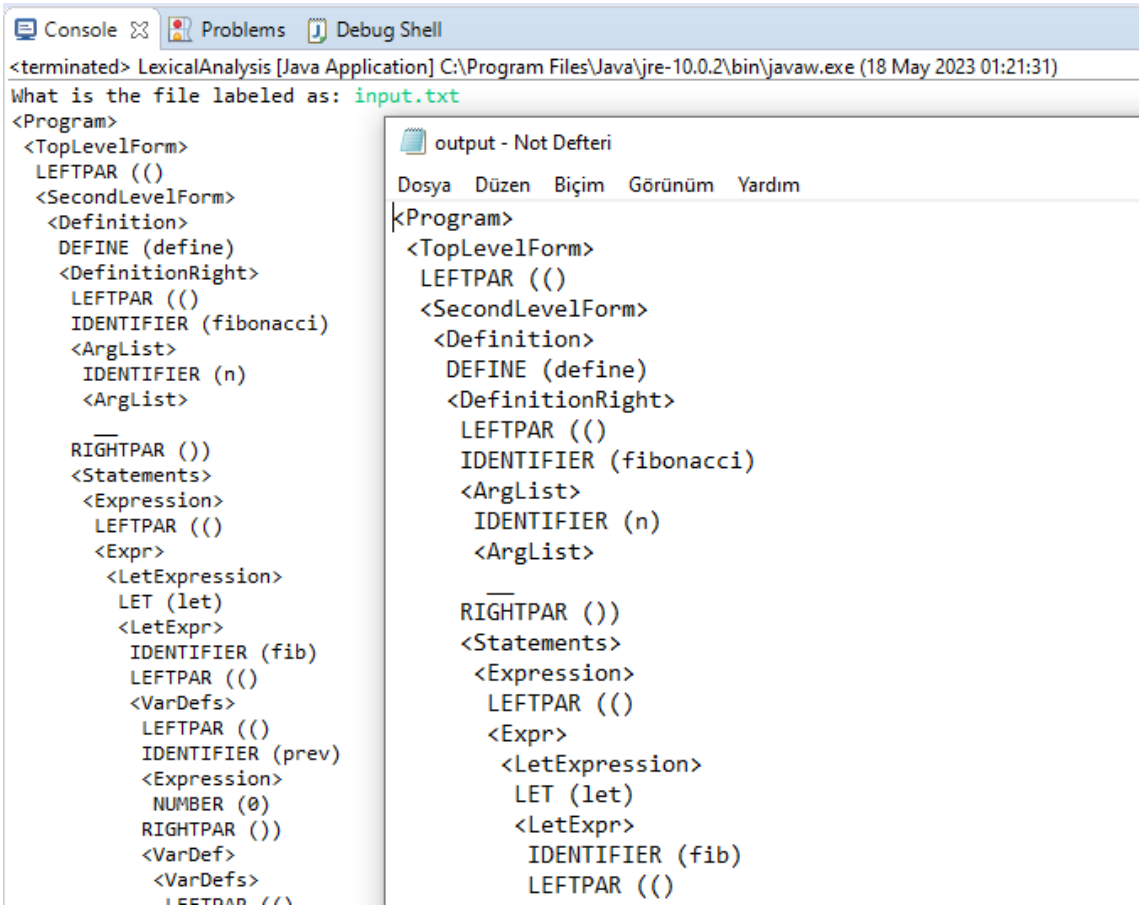


If the file has not been found, then you will get the output as follows:



If it is the case that the input has been found but there is an error, then the input and output will be as follows:

```
1 (define (fibonacci n)
2       ( let fib ((prev 0) (cur 1) (i 0))
3             ( if (= i n) cur (fib cur (+ prev cur) (+ i 1)))
4             (define junk n)))
```



Parts of The Assignment that are Completed

Everything works as desired in all our input test cases. (comments,parentheses, number literals, boolean literals, character literals, strings literals, keywords, identifiers) Also, we created the output of the Syntax Analyzer with the help of the previous part of the project.

Input/Output Examples

Example 1 (Correct Input)

```
(define (fibonacci n)
      ( let fib ((prev 0) (cur 1) (i 0))
            ( if (= i n) cur (fib cur (+ prev cur) (+ i 1)))))
```

<Program>
 <TopLevelForm>
  LEFTPAR (()
  <SecondLevelForm>
   <Definition>
    DEFINE (define)
    <DefinitionRight>
     LEFTPAR (()
     IDENTIFIER (fibonacci)
     <ArgList>
      IDENTIFIER (n)
      <ArgList>
       __
      RIGHTPAR ())
     <Statements>
      <Expression>
       LEFTPAR (()
       <Expr>
        <LetExpression>
         LET (let)
         <LetExpr>
          IDENTIFIER (fib)
          LEFTPAR (()
          <VarDefs>
           LEFTPAR (()
           IDENTIFIER (prev)
           <Expression>
            NUMBER (0)
           RIGHTPAR ())
           <VarDef>
            <VarDefs>
             LEFTPAR (()
             IDENTIFIER (cur)
             <Expression>
              NUMBER (1)
             RIGHTPAR ())
             <VarDef>
              <VarDefs>
               LEFTPAR (()
               IDENTIFIER (i)
               <Expression>
                NUMBER (0)

```
    RIGHTPAR ())
    <VarDef>

      __
RIGHTPAR ())
<Statements>
 <Expression>
  LEFTPAR (()
  <Expr>
   <IfExpression>
    IF (if)
    <Expression>
     LEFTPAR (()
     <Expr>
      <FunCall>
       IDENTIFIER (=)
       <Expressions>
        <Expression>
         IDENTIFIER (i)
         <Expressions>
          <Expression>
           IDENTIFIER (n)
           <Expressions>

            __
     RIGHTPAR ())
    <Expression>
     IDENTIFIER (cur)
    <EndExpression>
     <Expression>
      LEFTPAR (()
      <Expr>
       <FunCall>
        IDENTIFIER (fib)
        <Expressions>
         <Expression>
          IDENTIFIER (cur)
          <Expressions>
           <Expression>
            LEFTPAR (()
            <Expr>
             <FunCall>
              IDENTIFIER (+)
              <Expressions>
               <Expression>
                IDENTIFIER (prev)
                <Expressions>
                 <Expression>
                  IDENTIFIER (cur)
                  <Expressions>

                   __
            RIGHTPAR ())
           <Expressions>
            <Expression>
             LEFTPAR (()
             <Expr>
              <FunCall>
               IDENTIFIER (+)
               <Expressions>
                <Expression>
                 IDENTIFIER (i)
                 <Expressions>
```

<Expression>
NUMBER (1)
<Expressions>

RIGHTPAR ())
<Expressions>

RIGHTPAR ())
RIGHTPAR ())
RIGHTPAR ())
RIGHTPAR ())
<Program>

Example 2 (Incorrect Input)

```
1 (define (fibonacci n)
2         ( let fib ((prev 0) (cur 1) (i 0))
3               ( if (= i n) cur (fib cur (+ prev cur) (+ i 1)))
4               (define junk n)))
```

<Program>
<TopLevelForm>
LEFTPAR (()
<SecondLevelForm>
<Definition>
DEFINE (define)
<DefinitionRight>
LEFTPAR (()
IDENTIFIER (fibonacci)
<ArgList>
IDENTIFIER (n)
<ArgList>

RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<LetExpression>
LET (let)
<LetExpr>
IDENTIFIER (fib)
LEFTPAR (()
<VarDefs>
LEFTPAR (()
IDENTIFIER (prev)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (cur)
<Expression>
NUMBER (1)
RIGHTPAR ())
<VarDef>

```
    <VarDefs>
     LEFTPAR (()
     IDENTIFIER (i)
     <Expression>
      NUMBER (0)
     RIGHTPAR ())
     <VarDef>

    __
RIGHTPAR ())
<Statements>
 <Expression>
  LEFTPAR (()
  <Expr>
   <IfExpression>
    IF (if)
    <Expression>
     LEFTPAR (()
     <Expr>
      <FunCall>
       IDENTIFIER (=)
       <Expressions>
        <Expression>
         IDENTIFIER (i)
        <Expressions>
         <Expression>
          IDENTIFIER (n)
         <Expressions>

        __
     RIGHTPAR ())
    <Expression>
     IDENTIFIER (cur)
    <EndExpression>
     <Expression>
      LEFTPAR (()
      <Expr>
       <FunCall>
        IDENTIFIER (fib)
        <Expressions>
         <Expression>
          IDENTIFIER (cur)
         <Expressions>
          <Expression>
           LEFTPAR (()
           <Expr>
            <FunCall>
             IDENTIFIER (+)
             <Expressions>
              <Expression>
               IDENTIFIER (prev)
              <Expressions>
               <Expression>
                IDENTIFIER (cur)
               <Expressions>

              __
            RIGHTPAR ())
           <Expressions>
            <Expression>
             LEFTPAR (()
             <Expr>
              <FunCall>
```

```
                    IDENTIFIER (+)
                   <Expressions>
                    <Expression>
                     IDENTIFIER (i)
                    <Expressions>
                     <Expression>
                      NUMBER (1)
                     <Expressions>

                   __
                  RIGHTPAR ())
                 <Expressions>

                __
               RIGHTPAR ())
              RIGHTPAR ())
SYNTAX ERROR [4:14]: ')' is expected
```

Example 3 (Input with Lexical Error)

```
(define (0fibonacci n)
      ( let fib ((prev 0) (cur 1) (i 0))
            ( if (= i n) cur (fib cur (+ prev cur) (+ i 1)))))
```

```
LEFTPAR 1:1
DEFINE 1:2
LEFTPAR 1:9
LEXICAL ERROR [1:10]: Invalid token '0fibonacci'
```

Example 4 (File Not Found)