project

December 24, 2020

1 Final Project

Student Name: Simranjeet Singh

1.1 Submission

After answering all the questions, save your work in **Notebook** format file. Do not submit PDF files.

- Double-click on this cell
- Enter your name in the above placeholder, and evaluate this cell to reneder it correctly
- Save your work by pressing button in the toolbar
- Go to menu "File" -> "Download as"
- Select "Notebook (.ipynb)"
- Use downloaded file for Balckboard submission

For more information, see https://www.codecademy.com/articles/how-to-use-jupyter-notebooks

1.1.1 Coding Style

- Use functional F# style for writing your programs.
- Make sure that you do not use mutable variables & loops.
- Any imperative style programming is prohibited unless specified in the problem description.

For additional information of F# coding style see F# Style Guide.

1.1.2 Before You Submit

You are required to test that your submission works properly before submission. Make sure that your program compiles without errors. Once you have verified that the submission is correct, you can submit your work.

1.1.3 Your Submission

Program submissions should be done through the Blackboard.

1.1.4 F# Interactive Console

You can run your code in F# Interactive console inside Jupyter. The interactive console provides more relevant compilation information which might not be available in the notebooks cells.

For accessing F# Interactive console - Open a new terminal window, see https://youtu.be/IMdfXGHzz5g?t=840 - Type command `dotnet fsi` - Copy your code into the console for evaluation

1.2 Problem

In this project you write a recursive descent parser for a specific grammar.

1. Find your grammar definition in `grammars.md` file which corresponds to your ID.
2. Make appropriate changes to your grammar to convert it to the LL form if necessary
   - No left recursion
   - Pairwise disjoint productions
3. Write a recursive descent parser for your grammar. Your parser should output for every test sentence:
   - A test sentence itself
   - A list the grammar rules required to parse a correct sentence.
   - See a parsing example in rd-parser.ipynb script.
4. Use provided sentences from `grammars.md` file to test correctness of your parser.

You recursive-descent parser must output list of grammar rules required to parse a valid sentence of produced by your grammar. Several sentences are provided to you. If case of the error in parser input, i.e. invalid sentence, your parser need to output error and terminate execution.

1.3 Solution

Place your **LL grammar** in the following cell:

S -> eaf | eUT T -> e U -> aeV V -> cSV |

Place your **recursive descent parser** in the following cell:

[1]:
```
// Load some auxiliary tools
#load "grammartools.fsx"
open CSCI374.GrammarTools
open CSCI374.ParserTypes

type Tokenizer(grammar: PRODUCTION [], verbose: bool) =
    let mutable inputState = []
    let mutable curentToken = INVALID

    // Access to the
    member this.CurrentToken = curentToken
    member this.NextToken() =
        let tkn, input = CSCI374.Lexer.token inputState
        inputState <- input
        curentToken <- tkn
        this

    member this.InputState
        with set(str) = inputState <- Seq.toList str
```

```
    member this.IsVerbose = verbose
    member this.PrintRule ruleIdx =
        printGrammarRule false grammar ruleIdx // print rule
    new(grammar) = Tokenizer(grammar, false)

/// This infix operator function provides verbose output while calling
/// a particular production rule
let (==>) (cnxt:Tokenizer) (prod:Tokenizer->Tokenizer) =
    if cnxt.IsVerbose then
        printfn "Enter <%A> with token `%A`" prod cnxt.CurrentToken
    let nextcnxt = prod cnxt
    if cnxt.IsVerbose then
        printfn "Exit <%A> with token `%A`" prod cnxt.CurrentToken
    nextcnxt

/// This infix operator function will allow to print a production rule
/// call `cnxt @ 2` will print second grammar rule
let (@) (cnxt:Tokenizer) ruleIdx =
    cnxt.PrintRule ruleIdx
    cnxt

let grammar = parseGrammarString """
S -> eaf | eUT
T -> e
U -> aeV
V -> cSV |
"""
printfn "%A" grammar

// Show grammar rules
printGrammar grammar

let rec ProdS (cnxt:Tokenizer) =
    // check the current token is `E` then move to next token because S -> eaf␣
 ↪| eUT
    if cnxt.CurrentToken = E then
        cnxt.NextToken() |> ignore
        if cnxt.CurrentToken = A then
        // 1: S → eaf
         cnxt @(1)==> Match A ==> Match F
        else
            // 2: S → eUT
          cnxt @(2)==> Match E ==> ProdU ==> ProdT
    else
        cnxt
/// The function for production T → e is straight forward: match nonterminal `e`
and ProdT (cnxt:Tokenizer) =
```

```
    // 3: T -> e
    cnxt @(3)==> Match E

and ProdU (cnxt:Tokenizer) =
    // 4: U → aeV
    cnxt @(4)==> Match A ==> Match E ==> ProdV

and ProdV (cnxt:Tokenizer) =
    if cnxt.CurrentToken = C then
        //5: V → cSV
        cnxt.NextToken() @(5) ==> ProdS ==> ProdV
    else
        //6: V →
        cnxt @(6) ==> Match EPS

/// For each terminal symbol compare it with a current token
/// and if they match, continue with the next token, else there is an error
and Match term cnxt =
    if cnxt.IsVerbose then printfn "Match %A with %A" term cnxt.CurrentToken
    //printf "The Term `%A` and the current Token `%A`" term cnxt.CurrentToken
    // if we matched the current token with a terminal symbol
    if term = cnxt.CurrentToken then
        cnxt.NextToken() // read next token
    else
        failwith (sprintf "Cannot match symbol `%A` with `%A`" term cnxt.
 ↪CurrentToken)

/// Start parsing by calling starting symbol function
let parser (cnxt:Tokenizer) :Tokenizer =
    // Read token and pass it to the function for S rule
    cnxt.NextToken() ==> ProdS
```

```
S → eaf | eUT
T → e
U → aeV
V → cSV |
[|(S, [Terminal E; Terminal A; Terminal F]);
  (S, [Terminal E; NonTerminal U; NonTerminal T]); (T, [Terminal E]);
  (U, [Terminal A; Terminal E; NonTerminal V]);
  (V, [Terminal C; NonTerminal S; NonTerminal V]); (V, [Terminal EPS])|]
Grammar:
1: S → eaf
2: S → eUT
3: T → e
4: U → aeV
5: V → cSV
6: V →
```

Place your recursive recursive descent **parser tests** in the following cell:

[2]:
```
Tokenizer(grammar, true, InputState="eaeceafceaeceafee") |> parser |> ignore
```

```
Enter <<fun:it@1>> with token `E`
1: S → eaf
Enter <<fun:ProdS@58>> with token `A`
Match A with A
Exit <<fun:ProdS@58>> with token `E`
Enter <<fun:ProdS@58-1>> with token `E`
Match F with E
```

```
System.Exception: Cannot match symbol `F` with `E`
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0006.it@1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0006>.$FSI_0006.main@()
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0006.it@1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0006>.$FSI_0006.main@()
```

[3]:
```
Tokenizer(grammar, true, InputState="eaeceaeee") |> parser |> ignore
```

```
Enter <<fun:it@1-1>> with token `E`
1: S → eaf
Enter <<fun:ProdS@58>> with token `A`
Match A with A
Exit <<fun:ProdS@58>> with token `E`
Enter <<fun:ProdS@58-1>> with token `E`
Match F with E
```

```
System.Exception: Cannot match symbol `F` with `E`
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0007.it@1-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0007>.$FSI_0007.main@()
```

```
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0007.it@1-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0007>.$FSI_0007.main@()
```

[4]:
```
Tokenizer(grammar, true, InputState="eaeceaeeceafe") |> parser |> ignore
```

```
Enter <<fun:it@1-2>> with token `E`
1: S → eaf
Enter <<fun:ProdS@58>> with token `A`
Match A with A
Exit <<fun:ProdS@58>> with token `E`
Enter <<fun:ProdS@58-1>> with token `E`
Match F with E
```

```
System.Exception: Cannot match symbol `F` with `E`
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0008.it@1-2.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0008>.$FSI_0008.main@()
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0008.it@1-2.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0008>.$FSI_0008.main@()
```

[5]:
```
Tokenizer(grammar, true, InputState="eaeceafceaeeceafe") |> parser |> ignore
```

```
Enter <<fun:it@1-3>> with token `E`
1: S → eaf
Enter <<fun:ProdS@58>> with token `A`
Match A with A
Exit <<fun:ProdS@58>> with token `E`
Enter <<fun:ProdS@58-1>> with token `E`
Match F with E
```

```
System.Exception: Cannot match symbol `F` with `E`
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
```

```
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0009.it@1-3.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0009>.$FSI_0009.main@()
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0009.it@1-3.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0009>.$FSI_0009.main@()
```

[6]:
```
Tokenizer(grammar, true, InputState="eaeceafceaeee") |> parser |> ignore
```

```
Enter <<fun:it@1-4>> with token `E`
1: S → eaf
Enter <<fun:ProdS@58>> with token `A`
Match A with A
Exit <<fun:ProdS@58>> with token `E`
Enter <<fun:ProdS@58-1>> with token `E`
Match F with E
```

```
System.Exception: Cannot match symbol `F` with `E`
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0010.it@1-4.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0010>.$FSI_0010.main@()
   at FSI_0005.Match(TOKEN term, Tokenizer cnxt)
   at FSI_0005.ProdS@58-1.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at FSI_0005.ProdS(Tokenizer cnxt)
   at FSI_0010.it@1-4.Invoke(Tokenizer cnxt)
   at FSI_0005.op_EqualsEqualsGreater(Tokenizer cnxt, FSharpFunc`2 prod)
   at <StartupCode$FSI_0010>.$FSI_0010.main@()
```