Assignment 5

CSCI 3136: Principles of Programming Languages

Due Mar 25, 2019

Assignments are due on the due date before 23:59. All assignments must be submitted electronically via the course SVN server. Plagiarism in assignment answers will not be tolerated. By submitting your answers to this assignment, you declare that your answers are your original work and that you did not use any sources for its preparation other than the class notes, the textbook, and ones explicitly acknowledged in the answers. Any suspected act of plagiarism will be reported to the Faculty's Academic Integrity Officer and possibly to the Senate Discipline Committee. The penalty for academic dishonesty may range from failing the course to expulsion from the university, in accordance with Dalhousie University's regulations regarding academic integrity.

This is the second in a sequence of three programming assignments that culminate in an interpreter for Scheme--, a small subset of Scheme. This assignment focuses on implementing a recursive-descent parser (syntactic analyzer) for Scheme--.

Here is the grammar for a syntactically correct Scheme-- program. `Typewriter font` denotes literal text (which the scanner has transformed into a corresponding token). *Lowercase italic font* is a token name. Both are terminals. *CamelCase italic font* is a non-terminal. Review Assignment 3 for a list of tokens the scanner produces. Each of the following rules involving parentheses represents a whole group of rules where each matching pair of parentheses is replaced with matching pairs of round, square or curly parentheses. This is a result of allowing arbitrary parentheses to be used to delimit compound forms as in Scheme.

Program → ε | *TopLevelForm Program*
TopLevelForm → *Definition* | *FunCall*
Definition → `(` `define` *identifier Expression* `)`
| `(` `define` `(` *identifier ArgList* `)` *Statements* `)`
ArgList → ε | *identifier ArgList*
Statements → *Expression* | *Statement Statements*
Statement → *Definition* | *Expression*
Expressions → ε | *Expression Expressions*
Expression → *identifier* | *number* | *char* | *bool* | *string*
| *FunCall* | *Lambda*
| *QuotedExpression* | *BeginExpression* | *LetExpression* | *IfExpression* | *CondExpression*
FunCall → `(` *identifier Expressions* `)`
Lambda → `(` `lambda` `(` *ArgList* `)` *Statements* `)`
QuotedExpression → `(` `quote` *Expression* `)` | `'` *Expression*
BeginExpression → `(` `begin` *Statements* `)`
LetExpression → `(` `let` `(` *VarDefs* `)` *Statements* `)`
| `(` `let` *identifier* `(` *VarDefs* `)` *Statements* `)`
VarDefs → *VarDef* | *VarDef VarDefs*
VarDef → `(` *identifier Expression* `)`
IfExpression → `(` `if` *Expression Expression Expression* `)`
| `(` `if` *Expression Expression* `)`
CondExpression → `(` `cond` *CondBranches* `)`
CondBranches → *CondBranch* | *CondBranch CondBranches*
CondBranch → `(` *Expression Statements* `)`

Your task is to implement a recursive-descent parser that takes a Scheme-- program and prints its parse tree to `stdout` if the input is in fact a syntactically correct Scheme-- program. If the input is lexically incorrect, the parser should output the scanner's error message as in Assignment 3. If the input is lexically correct but syntactically incorrect, the parser should output a syntax error for the first invalid token. Here are two examples:

Correct input: On the following input

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

and using the above grammar (see "A Note on the Grammar" below), the parse tree your parser should print is shown on the next page. (The empty lines are not mistakes; they represent εs in the above grammar.)

Incorrect input: An example input of a lexically incorrect input was shown in Assignment 3. Such assignments should be treated exactly as in Assignment 3.

The following is an input that is lexically correct (all its components are valid tokens), but it is syntactically incorrect because the last statement in the `let`-block is not an expression:

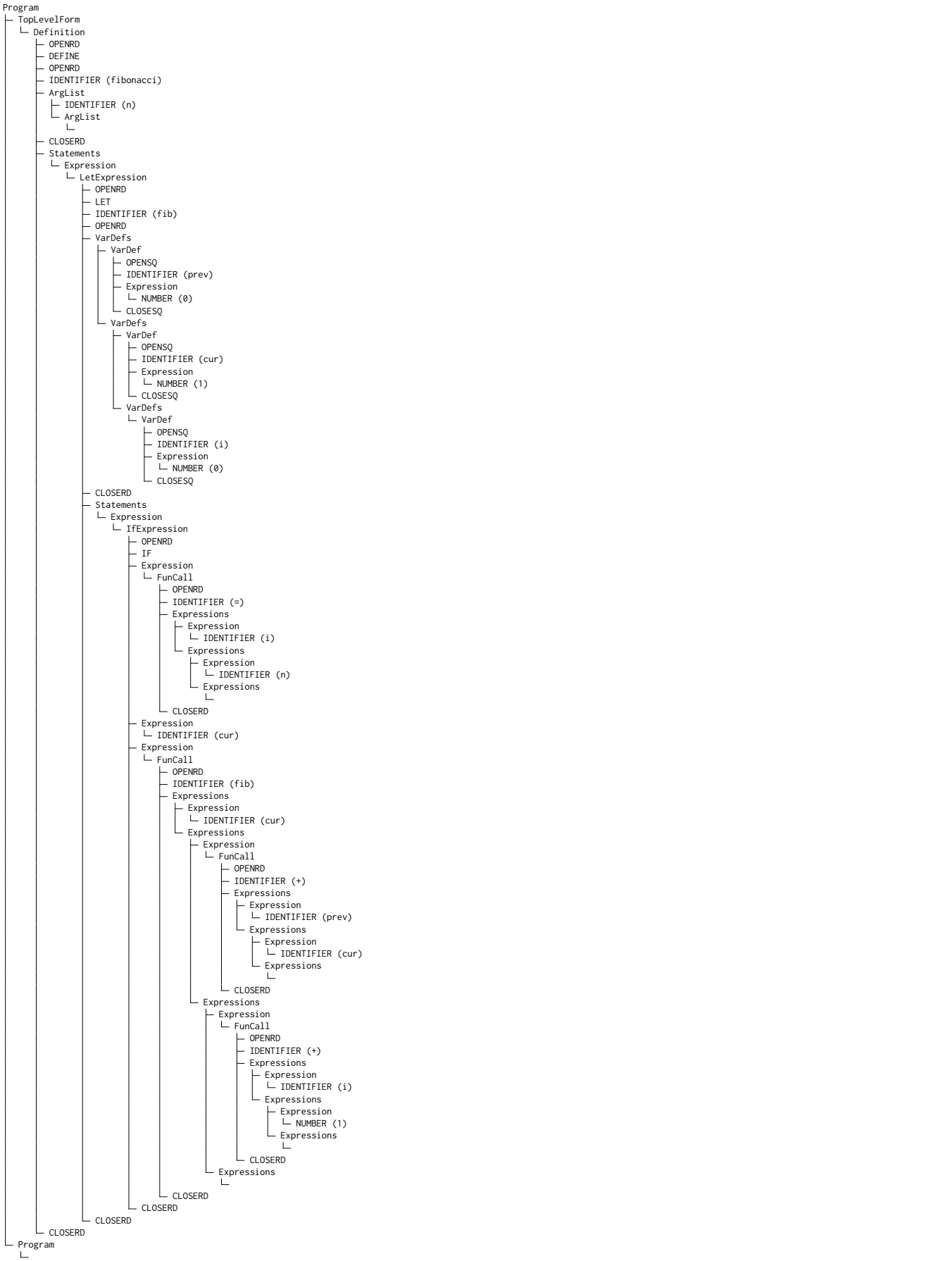
```
(define (fibonacci n)
  (let fib ([prev 0]
            [cur 1]
            [i 0])
    (if (= i n)
        cur
        (fib cur (+ prev cur) (+ i 1)))
    (define junk n)))
```

The parser recognizes that the last statement in the `let`-block is not an expression as soon as it sees the closing parenthesis of the `let`-block (the second-last parenthesis). It should declare this closing parenthesis as unexpected (because it did expect an expression, which never starts with a closing parenthesis):

```
SYNAX ERROR [8:20]: Unexpected token ')'
```

A Note on the Grammar

The grammar given here correctly describes the syntax of a Scheme-- program but is far from being LL(1). A first step in constructing your recursive-descent parser is to convert it into an LL(1) grammar. As part of your assignment, you must submit the grammar you constructed in a file `grammar.txt`. An example `grammar.txt` file showing the above grammer is provided at the end of this assignment for reference. The parse tree your parser outputs must match the rules of the grammar you constructed and will thus be different from the output shown in this assignment.


Program
TopLevelForm
Definition
OPENRD
DEFINE
OPENRD
IDENTIFIER (fibonacci)
ArgList
IDENTIFIER (n)
ArgList
CLOSERD
Statements
Expression
LetExpression
OPENRD
LET
IDENTIFIER (fib)
OPENRD
VarDefs
VarDef
OPENSQ
IDENTIFIER (prev)
Expression
NUMBER (0)
CLOSESQ
VarDefs
VarDef
OPENSQ
IDENTIFIER (cur)
Expression
NUMBER (1)
CLOSESQ
VarDefs
VarDef
OPENSQ
IDENTIFIER (i)
Expression
NUMBER (0)
CLOSESQ
CLOSERD
Statements
Expression
IfExpression
OPENRD
IF
Expression
FunCall
OPENRD
IDENTIFIER (=)
Expressions
Expression
IDENTIFIER (i)
Expressions
Expression
IDENTIFIER (n)
Expressions
CLOSERD
Expression
IDENTIFIER (cur)
Expression
FunCall
OPENRD
IDENTIFIER (fib)
Expressions
Expression
IDENTIFIER (cur)
Expressions
Expression
FunCall
OPENRD
IDENTIFIER (+)
Expressions
Expression
IDENTIFIER (prev)
Expressions
Expression
IDENTIFIER (cur)
Expressions
CLOSERD
Expressions
Expression
FunCall
OPENRD
IDENTIFIER (+)
Expressions
Expression
IDENTIFIER (i)
Expressions
Expression
NUMBER (1)
Expressions
CLOSERD
Expressions
CLOSERD
CLOSERD
CLOSERD
CLOSERD
Program

Submission Instructions

Place your code in a folder `a5` of your SVN repository and submit using the appropriate combination of `svn add` and `svn commit`.

Grammar in Plain Text Format

grammar.txt

```
Program          --> | TopLevelForm Program
TopLevelForm     --> Definition | FunCall
Definition       --> ( define identifier Expression )
                   | ( define ( identifier ArgList ) Statements )
ArgList          --> | identifier ArgList
Statements       --> Expression | Statement Statements
Statement        --> Definition | Expression
Expressions      --> | Expression Expressions
Expression       --> identifier | number | char | bool | string
                   | FunCall | Lambda
                   | QuotedExpression | BeginExpression | LetExpression
                   | IfExpression | CondExpression
FunCall          --> ( identifier Expression )
Lambda           --> ( lambda ( ArgList ) Statements )
QuotedExpression --> ( quote Expression ) | ' Expression
BeginExpression  --> ( begin Statements )
LetExpression    --> ( let ( VarDefs ) Statements )
                   | ( left identifier ( VarDefs ) Statements )
VarDefs          --> VarDef | VarDef VarDefs
VarDef           --> ( identifier Expression )
IfExpression     --> ( if Expression Expression Expression )
                   | ( if Expression Expression )
CondExpression   --> ( cond CondBranches )
CondBranches     --> CondBranch | CondBranch CondBranches
CondBranch       --> ( Expression Statements )
```