CS 314 Principles of Programming Languages

Project 2: LL(1) Parser in Scheme

In this project, you will implement a Scheme functions to generate the correct PREDICT sets for random, correctly-written, LL(1) grammars. Your functions will return the correct FIRST, FOLLOW, and PREDICT sets for the grammar given.

1 Structure for the Grammars and FIRST, FOLLOW & PREDICT Sets

This project will run functions based on random, correctly-generated LL(1) grammars. An LL(1) grammar will be represented by a list of lists, where each sub-list begins with a non-terminal symbol, then subsequently containing the right hand side of the rules corresponding to that non-terminal, which are lists comprising of terminal and/or nonterminal symbols. So, a grammar will be of the following form:

```
'(
(NT1 rhsrule11 rhsrule12 ...)
(NT2 rhsrule21 rhsrule22 ...)
...
(NTk rhsrulek1 rhsrulek2 ...)
)
```

We will use "eps" for the symbol ϵ. If the right hand side of the rule is the list '("eps"), then that rule corresponds to the symbol ϵ. **RULES CAN HAVE OTHER NON-TERMINALS IN THEM**. Remember, every non-terminal will have at least one production rule. The start nonterminal is the left hand side nonterminal of the first rule. In the case above, the start nonterminal is NT1.

The list of FIRST sets and FOLLOW sets have the following form and structure: a list of pairs consisting of the nonterminal and its corresponding FIRST or FOLLOW set:


```
'(
(NT1 FIRST(NT1))
(NT2 FIRST(NT2))
...
(NTk FIRST(NTk))
)
```


and

```
'(
(NT1 FOLLOW(NT1))
(NT2 FOLLOW(NT2))
...
(NTk FOLLOW(NTk))
)
```

The list of FIRST sets and FOLLOW sets are both associative lists. The list of PREDICT sets have the following form and structure: a list of pairs consisting of the rule (which is a list) and the PREDICT set for that rule:

```
'(
(rule11 PREDICT(rule11))
(rule12 PREDICT(rule12))
...
(rulek1 PREDICT(rulek1))
...
)
```

An example of a random LL(1) grammar is the following:

```
'(
(A (x B) ("eps"))
(B (y A) ("eps"))
)
```

which has the following BNF form:

```
A ::= xB | ϵ
B ::= yA | ϵ
```

where A and B are nonterminals. A is the start nonterminal. x and y are terminals, since they don't have any production rules. A correct output of the PREDICT sets for the grammar is as follows:

```
'(
((A ::= (x B)) (x))
((A ::= ("eps")) (eof)))
((B ::= (y A)) (y))
((B ::= ("eps")) (eof)))
)
```

The structure of the output is as follows: the number of elements in the list will be the number of non-terminals there are in the grammar. The first element of each element of the list is the non-terminal. Then, each subsequent element lists the rule, and then its PREDICT set. In this example, based on the grammar given above, if one is currently reducing A and encounters the symbol x, then x should be expanded with the rule $A ::= xB$, whereas if one is currently reducing B and encounters the symbol y, then y should be expanded with the rule $B ::= yA$. Note that in both cases, if one encounters EOF, (i.e. if one sees nothing else), then expansion of the rules is finished, and parsing is finished. This is denoted by '(eof).

1.1 FIRST Sets

Recall the method for constructing a FIRST set for a symbol (or collection of symbols) X:

1. If X is a terminal, then FIRST$(X) = \{X\}$.
2. If $X ::= \epsilon$, then $\epsilon \in$ FIRST(X).
3. For each X as a non-terminal, initialize FIRST$(X) = \{\}$.
4. Iterate until no more terminals or ϵ can be added to any FIRST(X), where $X = Y_1 Y_2 \dots Y_k$:
   (a) add a to FIRST(X) if $a \in$ FIRST(Y_1).
   (b) add a to FIRST(X) if $a \in$ FIRST(Y_j) and $\epsilon \in$ FIRST(Y_i), for all $1 \leq i < j$ and $j \geq 2$.
   (c) add ϵ to FIRST(X) if $\epsilon \in$ FIRST(Y_i), for all i where $1 \leq i \leq k$.

The algorithm is as follows, where NT is the set of all nonterminals in the grammar, and P is the set of production rules:

```
for all A ∈ NT:
    FIRST(A) = {}
while (FIRST sets are still changing) do
    for each p ∈ P, of the form X → Y1Y2...Yk do
        temp ← FIRST(Y1) - {ϵ}
        i ← 1
        while (i ≤ k - 1 and ϵ ∈ FIRST(Yi))
            temp ← temp ∪ (FIRST(Yi+1) - {ϵ})
            i ← i + 1
        end // while loop
        if i == k and ϵ ∈ FIRST(Yk)
            then temp ← temp ∪ {ϵ}
        end // if-then
        FIRST(X) ← FIRST(X) ∪ temp
    end // for loop
end // while loop
```

Also, refer to pg. 28 and 29 of Lecture 7 for more details.

1.2 FOLLOW Sets

Recall the algorithm for constructing a FOLLOW set for a non-terminal X:

1. Place EOF in FOLLOW(start nonterminal).
2. For all other nonterminals X, initialize FOLLOW(X) = {}.
3. Iterate until no more terminals can be added to any FOLLOW(X).
   (a) if a rule is of the form $A ::= \alpha B \beta$:
      i. if $\epsilon \in$ FIRST(β):
         A. FOLLOW(B) = FOLLOW(B) $\cup$ (FOLLOW(A) $\cup$ (FIRST(β) - {ϵ}))
      ii. otherwise:
         A. FOLLOW(B) = FIRST(β) $\cup$ FOLLOW(B).
   (b) if a rule is of the form $A ::= \alpha B$, then FOLLOW(B) = FOLLOW(B) $\cup$ FOLLOW(A).

The algorithm is as follows, where NT is the set of all nonterminals in the grammar, and P is the set of production rules:

```
for all A ∈ NT:
    FOLLOW(A) = {}
FOLLOW(start nonterminal) = {EOF}
while (FOLLOW sets are still changing) do
    for each p ∈ P, of the form A → B1B2...Bk do
        TRAILER ← FOLLOW(A)
        for i ← k down to 1
            if Bi ∈ NT then
                FOLLOW(Bi) ← FOLLOW(Bi) ∪ TRAILER
                if ϵ ∈ FIRST(Bi)
                    TRAILER ← TRAILER ∪ (FIRST(Bi) - {ϵ})
                else TRAILER ← FIRST(Bi)
            else TRAILER ← {Bi}
```


Also, refer to pg. 49 of Lecture 7.

1.3 PREDICT sets

Recall the algorithm for constructing a PREDICT for a rule $A ::= \delta$:

1. If $\epsilon \in$ FIRST(δ):
   (a) PREDICT($A ::= \delta$) = (FIRST(δ) - {ϵ}) $\cup$ FOLLOW(A)
2. otherwise:
   (a) PREDICT($A ::= \delta$) = FIRST(δ)

For more information regarding FIRST, FOLLOW & PREDICT sets, please refer to the slides for recitation week 3, the last two slides for recitation week 5, and lectures 7 and 8.

2 Project Description

For this project, you will have to complete the Scheme functions provided in order to generate the FIRST, FOLLOW, and PREDICT sets properly. You will need to complete the following tasks:

1. Complete the utility functions `getStartSymbol`, `getNonterminals`, `getProductions`, `updateAssocList`, and `getInitSets`.
   (a) Function signatures and descriptions:
      i.
      `(getStartSymbol grammar)` returns the start nonterminal of the grammar. As defined in the project description, the start nonterminal is the left-hand side nonterminal of the first rule of the grammar. (This function is useful for generating initial FOLLOW sets.)
      Input: grammar
      Output: the start non-terminal of the grammar.

ii.

`(getNonterminals grammar)` returns a list of all nonterminals in the grammar. **You may assume that every nonterminal in the grammar will have a rule!**
Input: grammar
Output: a list of all nonterminals in the grammar: `'(`NT_1 NT_2 ... NT_k`)`

iii.

`(getProductions grammar)` Returns a list of all production rules in the grammar. If a rule happens to look like "lhs ::= rhs1 | rhs2", getProductions will separate those two rules into different elements of the list.
Input: grammar
Output: a list of every rule in the grammar
Output format example:
Given an example grammar:

```
NT1 ::= rhs1 | rhs2
NT2 ::= rhs3
```

Which is of the form

```
'((NT1 (rhs1) (rhs2)) (NT2 (rhs3)))
```

getProductions should return

```
'((NT1 rhs1) (NT1 rhs2) (NT2 rhs3)).
```

iv.

`(updateAssocList assocList symbol set)` takes as input an associative list, a symbol, and the set to be union-ed with the corresponding list of the symbol.
Input: grammar

- assocList: associative list
- symbol: a symbol
- set: the set to be union-ed with the corresponding list of the symbol in assocList

Output: the updated associative list with the corresponding list of the symbol union-ed with set.
An associative list is defined as follows: given symbols a, b, ..., k and their corresponding lists $list_a$, $list_b$, ... $list_k$, an associative list is a list of pairs of symbols and their corresponding lists:

```
'(
(a list_a)
(b list_b)
...
(k list_k)
)
```

Each pair has the form `'(symbol list)`, where, for example, list can be the symbol's FIRST set or FOLLOW set. So, the command `(updateAssocList assocList b set)` with assocList being the associative list given above, will output the following associative list:

```
'(
(a list_a)
(b list_b ∪ set)
...
(k list_k)
)
```

v.

`(getInitSets NTs startSymbol setting)`: Depending on the setting (either 'first or 'follow), getInitSets returns the initialized FIRST sets or FOLLOW sets for all non-terminals in the grammar. **Inputs:**

- NTS: the list of all nonterminals in the grammar
- startSymbol: the start nonterminal of the grammar
- setting: either 'first or 'follow

Output: Depending on what the setting is, initSets returns an initialized list of empty FIRST sets or FOLLOW sets for each nonterminal of the grammar.
Output format example:
Given an example grammar:

```
NT1 ::= rhs1 | rhs2
NT2 ::= rhs3
```

Which is of the form

```
'((NT1 (rhs1) (rhs2)) (NT2 (rhs3)))
```

(getInitSets '(NT1 NT2) NT1 'first) returns

```
'((NT1 ()) (NT2 ()))
```

(getInitSets '(NT1 NT2) NT1 'follow) returns

```
'((NT1 (eof)) (NT2 ()))
```

2. Complete the functions `genFirstFunc`, `recurseFirstSets`, `genFollowFunc`, `recurseFollowSets`, `getFollowSets`, and `getPredictSets`.

   (a) Function signatures and descriptions:

   i. `(genFirstFunc NTs)` returns a function that computes FIRST(symbolSeq) given a sequence of symbols (terminals and nonterminals) symbolSeq and an initial list of FIRST sets for all nonterminals in the grammar.
   Input: NTs: the list of all nonterminals in the grammar
   Output: a function called first that takes as input a sequence of symbols symbolSeq, and an initial list of FIRST sets firstSets, and outputs FIRST(symbolSeq). first has the following inputs and outputs:

   - **Inputs:**
     - symbolSeq: sequence of symbols
     - firstSets: a list of (potentially unfinished) FIRST sets for all nonterminals in the grammar
   - **Output:** FIRST(symbolSeq), i.e. the application `(first symbolSeq firstSets)` given inputs symbolSeq and firstSets.

   ii. `(recurseFirstSets rules firstSets firstFunc)` goes through each rule in the grammar and updates the FIRST sets based on the current rule.
   Inputs:

   - rules: the list of all production rules in the grammar
   - firstSets: a list of (potentially unfinished) FIRST sets
   - firstFunc: a function that takes as input a sequence of symbols and list of FIRST sets, and returns FIRST(sequence of symbols). (genFirstFunc generates this argument.)

   Output: an updated firstSets after making one pass through all the rules in the grammar

   iii. `(genFollowFunc NTs)` returns a function that computes FOLLOW(symbolSeq) given a sequence of symbols (terminals and nonterminals) and an initial list of FOLLOW sets for all nonterminals in the grammar.
   Input: NTs: the list of all nonterminals in the grammar
   Output: a function follow that takes as input a sequence of symbols symbolSeq, and an initial list of FOLLOW sets, the variable trailer, and the list of COMPLETED FIRST sets firstSets, and outputs FOLLOW(symbolSeq). follow has the following inputs and output:

   - **Inputs:**
     - symbolSeq: sequence of symbols
     - followSets: a list of (potentially unfinished) FOLLOW sets for all nonterminals in the grammar
     - trailer: trailer variable (Check the algorithm on FOLLOW sets)
     - firstSets: list of COMPLETED FIRST sets for all nonterminals in the grammar
   - **Output:** FOLLOW(symbolSeq), i.e. the application `(follow symbolSeq followSets trailer firstSets)` given inputs symbolSeq, followSets, trailer, and firstSets.

   iv. `(recurseFollowSets rules followSets followFunc)` goes through each rule in the grammar and updates the FOLLOW sets based on the current rule.
   Inputs:

- rules: a list of production rules in the grammar
- followSets: a list of (potentially unfinished) FOLLOW sets
- followFunc: a function that takes as input a sequence of symbols and list of FOLLOW sets, and returns FOLLOW(sequence of symbols). (genFollowFunc generates this argument.)

Output: an updated followSets after making one pass through all the rules in the grammar

v. `(getFollowSets rules followSets followFunc)` returns the FOLLOW sets of all nonterminals in the grammar if the FOLLOW sets had no change.

Inputs:

- rules: a list of production rules in the grammar
- followSets: a list of (potentially unfinished) FOLLOW sets
- followFunc: a function that takes as input a sequence of symbols and list of FOLLOW sets, and returns FOLLOW(sequence of symbols).

Output: the updated followSets, which is a list of the FOLLOW sets of every non-terminal in the grammar

vi. `(getPredictSets rules NTs firstSets followSets firstFunc)` returns the PREDICT sets for each rule in the grammar

Inputs:

- rules: the list of production rules of the grammar
- NTs: the list of all nonterminals in the grammar
- firstSets: the FIRST sets of all nonterminals in the grammar
- followSets: the FOLLOW sets of all nonterminals in the grammar
- firstFunc: a function that takes as input a sequence of symbols and list of FIRST sets, and returns FIRST(sequence of symbols)

Output: a list of pairs, one for each production rule in the grammar, where the first element is the production rule output as a list, and the second element is the PREDICT set for that production rule.

Output format example: Given an example grammar:

```
A ::= xB | "eps"
B ::= yA | "eps"
```

Which is of the form

```
'((A (x B) ()) (B (y A) ()))
```

getPredictSets should return

```
'(
  ((A ::= (x B)) (x))
  ((A ::= ("eps")) (eof))
  ((B ::= (y A)) (y))
  ((B ::= ("eps")) (eof)))
```

You MUST use let* for this problem: How do you define A? How do you define delta? How do you define FIRST(delta)?

2.1 Function closure

`genFirstFunc` and `genFollowFunc` requires you to return a function, rather than a list or numerical value. If you bind certain variables inside the closure (either using let, let*, letrec, or lambda), you can pre-load those bound variables when defining the function you wish to return. For example, given the function `plusXY`:

```
(define plusXY
    (lambda (x y) (+ x y)))
```

a way to return a function `plusX` that is similar to `plusXY` with y pre-loaded can be as follows:

```
(define plusX
    (lambda (y)
        (let ((plusXFunc (lambda (x) (+ x y))))
            plusXFunc)))
```

Then, (plusX 2) will return the function (lambda (x) (+ x 2)). For example, ((plusX 2) 10) = 12.

2.2 Given Functions

For this project, most of the functions are self-contained, so there should be no need to construct new helper functions. If you decide to use helper functions that you construct, please detail clearly what the function does using comments. Remember to use the semi-colon for comments. Here is a list of the given utility functions (in `utilFuncs.ss`) and useful built-in Scheme functions:

1. `(contains? lst x)` checks whether or not an element x is in the list lst. (This is a top-level check: meaning `(contains '(1 2 3) 2)` returns #t but `(contains '(1 (2 3)) 2)` returns #f.)
2. `(union a b)` returns a new list that is a ∪ b, where a and b are sets. Since a set is defined to be a list where no element appears twice, the new list should also contain only elements of a and b such that there are no duplicates.
3. `(removeMatch lst x)` returns a new list with every instance of x removed from the list lst. If x is not in lst at all, removeMatch just returns the original list lst with no changes.
4. `(epsilon)` defines ϵ as "eps". You can call this function without arguments like this: (epsilon). It will return the symbol "eps".

In addition to these functions, you are also given getFirstSets.

- `(getFirstSets rules firstSets firstFunc)` returns the FIRST sets of all nonterminals in the grammar if the FIRST sets had no change.

  Inputs:

  - rules: the list of all production rules in the grammar
  - firstSets: a list of (potentially unfinished) FIRST sets
  - firstFunc: a function that takes as input a sequence of symbols symbolSeq and list of FIRST sets, and returns FIRST(symbolSeq). (genFirstFunc generates this argument.)

  Output: the updated firstSets, which is a list of the FIRST sets of every non-terminal in the grammar

3 Implementation & Useful Built-in Scheme Functions

For the implementation of the required functions, you can use standard built-in Scheme functions such as `append`, `list`, `let`, `let*`, `letrec`, `map`, `assoc`, and `reverse`. Here are some useful built-in Scheme functions:

1. `(map function list)` is a useful built-in Scheme function that returns an updated list, where each element of the list is applied with function. For example: `(map even? '(1 2 3 4 5))` = '(#f #t #f #t #f).
2. `(assoc assocList symbol)` is a useful built-in Scheme function. An associative list is a list of pairs consisting of a symbol and a list corresponding to that symbol. An associative list has the following form.

   '(
   (a List_a)
   (b List_b)
   ...
   (k List_k)
)

   (assoc symbol assocList) takes as input an associative list assocList, and a symbol, and returns the symbol and corresponding list of the symbol as a list. In this case, given the above associative list as assocList, (assoc 'a assocList) returns '(a List_a).
3. `(reverse lst)` is a useful built-in Scheme function that returns the reverse of a list lst. For example: `(reverse '(1 (2 3) 4 (5 6 7)))` = '((5 6 7) 4 (2 3) 1).

Some utility functions will be provided for you in `utilFuncs.ss`. You will use each of `let`, `let*`, `letrec` at least once. **You must not use assignment (set!) in Scheme**.

4 How To Get Started

Copy the files `proj2.ss` and `utilFuncs.ss` into your own subdirectory. Both files must be in the same subdirectory. Open `proj2.ss` using either DrRacket or racket from the terminal (check Recitation week 7 for more information on how to do this). **Remember to set the specific language to be R5RS.** The file `proj2.ss` is split into five sections: The first section contains the utility functions, the next three sections contains the FIRST, FOLLOW, PREDICT set generators, and the last section contains the example grammars and the tester functions.

Fill in the functions where it says "YOUR CODE GOES HERE". Once you have filled in the functions with your own Scheme code, run `proj2.ss`.

The tester functions pre-define the variables and outputs the PREDICT sets for each grammar. In order to swap out grammars, replace "grammar0" with the appropriate numbered grammar in the line `(define grammar grammar<num>)`. In the last section of the project description, we provide the FIRST, FOLLOW, and PREDICT sets for the four example grammars given to you.

In addition to the tester functions and the example LL(1) grammars, we have also provided commands to generate FIRST and FOLLOW sets separately. These commands are included in section 5 of `proj2.ss`. **REMEMBER TO COMMENT OUT THE TEST COMMANDS BEFORE SUBMITTING YOUR FILE.** In another words, your `proj2.ss` should print nothing.

Read the `proj2.ss` already given to you, and understand the structure of the code. Use similar techniques to assist you in constructing the required functions.

5 Grading

You will submit your modified version of the file `proj2.ss` via Sakai. Your programs will be graded primarily on functionality, i.e. whether you have printed the correct PREDICT sets or not for each rule. Note that for grading purposes, you have to print each set with the correct elements, i.e. the order of elements in each set does not matter. This will be verified through automatic testing on a list of 10 correct LL(1) grammars, 4 of which have been given to you as example grammars. You do not have to re-submit `utilFuncs.ss`.

6 Questions

Remember, if you have any questions regarding this project, the Sakai forum has a section provided for questions regarding Project 2. **START EARLY**, since you will run into issues that will need to be resolved on the way.

7 FIRST, FOLLOW, & PREDICT sets for the Example Grammars

There are four grammars given to you in section 5 of `proj2.ss`. This section lists their FIRST, FOLLOW, and PREDICT sets, in the form that would be returned by `proj2.ss`. Note that the order of elements in the FIRST, FOLLOW, and PREDICT sets do not matter. For example, for grammar1, FIRST(a) could've been '(x "eps") instead of '("eps" x). Also, note that spaces between the terminals and nonterminals do not matter. Recall the structure and form of the list of FIRST, FOLLOW, and PREDICT sets. (Refer to Section 1 of this project description for the form and structure of the FIRST, FOLLOW, and PREDICT sets.)

- grammar0

  ```
  start ::= a
  ```

  FIRST sets

  ```
  '((start (a)))
  ```

  FOLLOW sets

  ```
  '((start (eof)))
  ```

  PREDICT sets

  ```
  '(
    ((start ::= (a)) (a))
   )
  ```

- grammar1

```
A ::= xB | ϵ
B ::= yA | ϵ
```

FIRST sets

```
'((a ("eps" x))
  (b ("eps" y)))
```

FOLLOW sets

```
'((a (eof))
  (b (eof)))
```

PREDICT sets

```
'(
  ((a ::= (x b)) (x))
  ((a ::= ("eps")) (eof))
  ((b ::= (y a)) (y))
  ((b ::= ("eps")) (eof))
 )
```

- grammar2

```
start ::= expr
expr  ::= + term term
term  ::= a | b | c
```


FIRST sets

```
'(
  (start ("+"))
  (expr ("+"))
  (term (c b a))
 )
```

FOLLOW sets

```
'((start (eof))
  (expr (eof))
  (term (a b c eof))
 )
```

PREDICT sets

```
'(
  ((start ::= (expr)) ("+"))
  ((expr ::= ("+" term term)) ("+"))
  ((term ::= (a)) (a))
  ((term ::= (b)) (b))
  ((term ::= (c)) (c))
 )
```

- grammar3

```
start     ::= stmts
stmts     ::= assgn morestmts
morestmts ::= , stmts | ϵ
assgn     ::= var = value
var       ::= a | b | c | d | e
value     ::= 0 | 1 | 2 | 3 | ... | 9
```

FIRST sets

```
'(
  (start (e d c b a))
  (stmts (e d c b a))
  (morestmts ("eps" ","))
  (assgn (e d c b a))
  (var (e d c b a))
  (value (9 8 7 6 5 4 3 2 1 0))
 )
```

FOLLOW sets

```
'(
  (start (eof))
  (stmts (eof))
  (morestmts (eof))
  (assgn ("," eof))
  (var ("="))
  (value ("," eof))
 )
```

PREDICT sets

```
'(
  ((start ::= (stmts)) (e d c b a))
  ((stmts ::= (assgn morestmts)) (e d c b a))
  ((morestmts ::= ("," stmts)) (","))
  ((morestmts ::= ("eps")) (eof))
  ((assgn ::= (var "=" value)) (e d c b a))
  ((var ::= (a)) (a))
  ((var ::= (b)) (b))
  ((var ::= (c)) (c))
  ((var ::= (d)) (d))
  ((var ::= (e)) (e))
  ((value ::= (0)) (0))
  ((value ::= (1)) (1))
  ((value ::= (2)) (2))
  ((value ::= (3)) (3))
  ((value ::= (4)) (4))
  ((value ::= (5)) (5))
  ((value ::= (6)) (6))
  ((value ::= (7)) (7))
  ((value ::= (8)) (8))
  ((value ::= (9)) (9))
 )
```