CS 314 Principles of Programming Languages

Project 2: LL(1) Parser Generator in OCaml

In this project, you will implement an LL(1) parser generator using OCaml. Specifically you need to implement functions that generate FIRST, FOLLOW, and PREDICT sets for random, correctly-written, LL(1) grammar. Based on these set functions, you will then implement a parser generator function that returns a parser for identifying correct/incorrect input. An EXTRA-CREDIT component is available for generating parse trees from any correct input string. This is NOT a group project. You must work on your own.

1 Data Structures for Grammars and Sets

In this project, a grammar is represented by a tuple consisting of the start symbol (a string) and a list of rules. Each rule is represented by a tuple consisting of the LHS (a string) and the RHS (a list of strings). Thus the type of a grammar is `string * (string * string list) list`. For example, consider the following simple grammar:

$$S \to aSb \mid \epsilon$$

In this project, it is represented by `("S",[("S",["a";"S";"b"]);("S",[])])`

We use the `Set` module from the standard library to represent sets of symbols. We define SymbolSet = Set.Make(String). Thus FIRST, FOLLOW, and PREDICT sets all have type SymbolSet.t.

If s has type SymbolSet, and x is a symbol, then (SymbolSet.mem x s) checks whether x is in s. See OCaml manual on how to work with `Set`: https://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.html

There are two special symbols that may appear in FIRST and FOLLOW sets. We use the string "eps" to represent the ϵ symbol that occurs in FIRST sets. We use the string "eof" to represent the end-of-input marker that occurs in FOLLOW sets. It is guaranteed that these two strings never occur as terminal or non-terminal symbols in any input grammar test cases.

We use the `Map` module from the standard library to manipulate associative maps from non-terminal symbols to their FIRST and FOLLOW sets. Thus we define SMap = Map.Make(String). We define the type of the map using type symbolMap = SymbolSet.t SMap.t, such that each key in the associative map is a string, and each key is associated with a SymbolSet. If m is a symbol map, containing the FIRST sets of non-terminals, and x is a non-terminal symbol, then (SMap.find x m) returns the FIRST set of x in m. You can also use other functions in the Map module, such as SMap.add for inserting into the map and SMap.find for searching in the map. See OCaml manual on how to work with Map: https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html

The PREDICT sets are organized as a list of tuples. Each tuple consists of a rule (LHS and RHS) and a symbol set (the PREDICT set). Thus the list of PREDICT sets has type `((string * string list) * SymbolSet.t) list`.

2 Algorithms for Constructing FIRST, FOLLOW, and PREDICT Sets

2.1 FIRST Sets

FIRST sets are defined not only for every non-terminal symbol, but more generally for every sequence of symbols. If $Y = Y_1 Y_2 \cdots Y_k$ is a sequence of symbols, then FIRST(Y) can be computed in the following way:

1. If $Y = \epsilon$, then FIRST$(Y) = \{\epsilon\}$.
2. If Y_1 is a terminal, then FIRST$(Y) = \{Y_1\}$.
3. If Y_1 is a non-terminal, then:
   (a) If $\epsilon \in$ FIRST(Y_1), then FIRST$(Y) = ($FIRST$(Y_1) - \{\epsilon\}) \cup$ FIRST$(Y_2 \cdots Y_k)$
   (b) If $\epsilon \notin$ FIRST(Y_1), then FIRST$(Y) =$ FIRST(Y_1).

Recall the algorithm to construct FIRST sets for a grammar:

1. For each non-terminal X, initialize FIRST$(X) = \{\}$.
2. Iterate until no more terminals or ϵ can be added to any FIRST(X), where X is a non-terminal:
   For each rule of the form $X ::= Y$ where Y is a sequence of symbols, add FIRST(Y) to FIRST(X).

2.2 FOLLOW Sets

Recall the algorithm for constructing FOLLOW sets for a grammar:

1. For each non-terminal X, initialize FOLLOW(X) = {}.
2. Add the end-of-input marker EOF to FOLLOW(S) where S is the start symbol.
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):
         FOLLOW(B) $\leftarrow$ FOLLOW(B) $\cup$ (FOLLOW(A) $\cup$ (FIRST(β) - {ϵ}))
      ii. otherwise:
         FOLLOW(B) $\leftarrow$ FIRST(β) $\cup$ FOLLOW(B).
   (b) if a rule is of the form $A ::= \alpha B$, then FOLLOW(B) $\leftarrow$ FOLLOW(B) $\cup$ FOLLOW(A).

The pseudo code below demonstrates the idea of how to the follow sets, where P is the set of production rules and NT is the set of non-terminals. Note that the pseudocode is only for illustration purpose (it uses loops in imperative languages). You will need to implement this as recursive functions, not using loops.

```
while (FOLLOW sets are still changing) do
    for each p $\in$ P, of the form A $\rightarrow$ $B_1 B_2 ... B_k$ do
        for i $\leftarrow$ 1 to k - 1
            if $B_i$ $\in$ NT then
                if $\epsilon$ $\in$ FIRST($B_{i+1} ... B_k$) then
                    FOLLOW($B_i$) $\leftarrow$ FOLLOW($B_i$) $\cup$ (FIRST($B_{i+1} ... B_k$) - {$\epsilon$}) $\cup$ FOLLOW(A)
                else
                    FOLLOW($B_i$) $\leftarrow$ FOLLOW($B_i$) $\cup$ FIRST($B_{i+1} ... B_k$)
        if $B_k$ $\in$ NT then
            FOLLOW($B_k$) $\leftarrow$ FOLLOW($B_k$) $\cup$ FOLLOW(A)
```

2.3 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 on FIRST, FOLLOW, and PREDICT sets, refer to the notes of lectures 5–7 and recitations week 3–4, and *Programming Language Pragmatics* Section 2.3.

3 Project Specification

You are given the following files:

1. `proj2_types.ml` – This file contains the type definitions for use in this project. All other files depends on this file. DO NOT modify this file.
2. `proj2_types.mli` – This file contains the signature of proj2_types.ml. DO NOT modify this file.
3. `proj2.ml` – This file contains stub functions which you must implement. This section will specify each of these functions.
4. `proj2.mli` – This file contains the signature of each function in `proj2.ml`. DO NOT modify this file. Your implementation will be checked against this file, so you should not change the signature of any existing functions. However, you are allowed to add helper functions as you need.

5. `proj2_driver.ml` – This file contains some functions that can help you debug your `proj2.ml`. See Section 4 of this document on how to use this file.

6. `Makefile` – This file specifies the dependence between different files and commands for generating object files and testing/submitting your code. See Section 4 of this document on how to use this file.

7. `codetest.ml` – This file provides example code on how to test the functions you generated. See section 4 of this document on how to use this file.

3.1 Conventions

We adopt the following conventions when specifying the functions that you will need to implement:

1. We use `simple_grammar` to represent a simple LL(1) grammar.
   `simple_grammar = ("S",[("S",["a";"S";"b"]);("S",[])])`

2. We use `complex_grammar` to represent a slightly more complex grammar for the same language.
   `complex_grammar = ("S",[("S",["T"]);("T",["a";"S";"b"]);("T",[])])`

3. We use {`"a","b","c"`} to represent a symbol set that contains `"a","b","c"`. This notation is NOT recognized by OCaml. Instead you must build sets using `SymbolSet.empty` and `SymbolSet.add`.

4. We use `[a -> x;b -> y;c -> z]` to represent an associative map that maps `a` to `x`, `b` to `y`, and `c` to `z`. This notation is NOT recognized by OCaml. Instead you must build maps using `SMap.empty` and `SMap.add`.

5. In `proj2.ml`, some functions are declared with the `rec` keyword and some are not. This is only a hint, not a rule about whether the function should be recursive. In other words you can always add or remove the `rec` keyword.

3.2 Function getStartSymbol

`getStartSymbol (g : grammar) : string`

This function returns the start symbol of `g`.

Example: `getStartSymbol simple_grammar` → `"S"`

3.3 Function getNonterminals

`getNonterminals (g : grammar) : string list`

This function returns a list containing all non-terminals symbols in `g`. It is guaranteed that every non-terminal symbol appears as the LHS of some rule. The returned list does not need to be sorted in any way. You do not need to remove duplicate entries.

Example: `getNonterminals simple_grammar` → `["S";"S"]` or `["S"]`

3.4 Function getInitFirstSets

`getInitFirstSets (g : grammar) : symbolMap`

This function build an initial, empty map from non-terminals to FIRST sets. The returned symbol map should satisfy:

1. The keys in the map are exactly the non-terminals of `g`.

2. The value associated with each key is an empty set {}.

Example: `getInitFirstSets simple_grammar` → `["S" -> {}]`

Note: this function should ONLY return first sets for non-terminals (not including that for terminals).

3.5 Function getInitFollowSets

`getInitFollowSets (g : grammar) : symbolMap`

This function builds an initial, empty map from non-terminals to FOLLOW sets. The returned symbol map should satisfy:

1. The keys in the map are exactly the non-terminals of `g`.
2. The value associated with the start symbol is {`"eof"`}.
3. The value associated with every other non-terminal is {}.

Example: `getInitFollowSets simple_grammar` → `["S" -> {"eof"}]`

3.6 Function computeFirstSet

`computeFirstSet (first : symbolMap) (symbolSeq : string list) : SymbolSet.t`

This function takes a (possibly incomplete) map of FIRST sets, `first`, and a sequence of symbols, `symbolSeq`, and returns the FIRST set of `symbolSeq`, according to the given FIRST sets. The sequence of symbols `symbolSeq` has type `string list`. For instance symbolSeq = [”A”;”B”] represents the sequence AB for first set calculation.

Example:

```
computeFirstSet ["A" -> {"a","b"};"B" -> {"c","eps"}] ["A";"B"] → {"a","b"}
computeFirstSet ["A" -> {"a","eps"};"B" -> {"b","eps"}] ["A";"B"] → {"a","b","eps"}
```

3.7 Function recurseFirstSets

`recurseFirstSets (g : grammar) (first : symbolMap) firstFunc : symbolMap`

The parameter `firstFunc` is a function that returns the first set for a given sequence of symbols. It has the same signature and functionality as `computeFirstSet` defined above.

This function takes a grammar `g`, a (possibly incomplete) map of FIRST sets - `first`, and the function for computing first set of a sequence `firstFunc`, executes **one** iteration of the FIRST set construction algorithm (see section 2.1 of this document). That is, it iterates over the rules of the grammar in the listed order. For each rule, it adds FIRST(RHS) to FIRST(LHS), and moves on to the next rule. It returns the updated map of FIRST sets after processing each rule once.

Example:

```
recurseFirstSets simple_grammar (getInitFirstSets simple_grammar) computeFirstSet
→ ["S" -> {"a","eps"}]
```

```
recurseFirstSets complex_grammar (getInitFirstSets complex_grammar) computeFirstSet
→ ["S" -> {};"T" -> {"a","eps"}]
```

Note: The implementation should NOT call `computeFirstSet` directly, but use `firstFunc` to compute the FIRST set of the RHS of each rule. The reason is that, if your `computeFirstSet` implementation is incorrect, we might give you partial points by testing your `recurseFirstSets` function with our correct version of `computeFirstSet`.

3.8 Function getFirstSets

`getFirstSets (g : grammar) (first : symbolMap) firstFunc : symbolMap`

This function takes a grammar, `g`, a (possibly incomplete) map of FIRST sets, `first`, and an implementation of `computeFirstSet`, `firstFunc`. It should call `recurseFirstSets` repeatedly to update the map, until it no longer changes. At this point the map becomes a complete map of FIRST sets, which the function returns.

Example: See provided testcases.

3.9 Function updateFollowSet

`updateFollowSet (first : symbolMap) (follow : symbolMap) (nt : string) (symbolSeq : string list) : symbolMap`

This function takes a completed map of FIRST sets, `first`, a (possibly incomplete) map of FOLLOW sets, `follow`, a non-terminal symbol, `nt`, and a sequence of symbols, `symbolSeq`. Here `nt` and `symbolSeq` are the LHS and RHS of a production rule. This function executes line 3–8 of the pseudocode in section 2.2, and returns the updated FOLLOW map.

Example:
`updateFollowSet (getFirstSets simple_grammar (getInitFirstSets simple_grammar) computeFirstSet) (getInitFollowSets simple_grammar) "S" ["a";"S";"b"]` → `["S" -> {"b","eof"}]`

3.10 Function recurseFollowSets

`recurseFollowSets (g : grammar) (first : symbolMap) (follow : symbolMap) followFunc : symbolMap`

`followFunc` is a function that has the same signature and functionality as `updateFollowSet`.

This function takes a grammar, `g`, a complete map of FIRST sets, `first`, a (possibly incomplete) map of FOLLOW sets, `follow`, and an implementation of `updateFollowSet`, and executes **one** iteration of the FOLLOW set construction algorithm (see section 2.2). That is, it iterates over the rules of the grammar in the listed order. For each rule, it invokes `followFunc` to update the FOLLOW map, before moving on to the next rule. It returns the updated FOLLOW map after processing each rule once.

Example:
`recurseFollowSets simple_grammar (getFirstSets simple_grammar (getInitFirstSets simple_grammar) computeFirstSet) (getInitFollowSets simple_grammar) updateFollowSet`
→ `["S" -> {"b","eof"}]`

`recurseFollowSets complex_grammar (getFirstSets complex_grammar (getInitFirstSets complex_grammar) computeFirstSet) (getInitFollowSets complex_grammar) updateFollowSet`
→ `["S" -> {"b","eof"};"T" -> {"eof"}]`

Note: The implementation should NOT call `updateFollowSet` directly, but use `followFunc` to update the FOLLOW map. The reason is that, if your `updateFollowSet` implementation is incorrect, we might give you partial points by testing your `recurseFollowSets` function with our correct version of `updateFollowSet`.

3.11 Function getFollowSets

`getFollowSets (g : grammar) (first : symbolMap) (follow : symbolMap) followFunc : symbolMap`

This function takes a grammar, `g`, a complete map of FIRST sets, `first`, a (possibly incomplete) map of FOLLOW sets, `follow`, and an implementation of `updateFollowSet`. It should call `recurseFollowSets` repeatedly to update the FOLLOW map until it no longer changes. At this point we have a complete FOLLOW map, which the function returns.

Example: See provided testcases.

3.12 Function getPredictSets

`getPredictSets (g : grammar) (first : symbolMap) (follow : symbolMap) firstFunc : ((string * string list) * SymbolSet.t) list`

This function takes a grammar, `g` and completed maps of FIRST and FOLLOW sets, `first` and `follow`, and constructs the PREDICT set for each production rule.

Example: See provided testcases.

3.13 Function tryDerive

`tryDerive (g : grammar) (input : string list) : bool`

In this function you will put everything you wrote together, to implement an LL(1) parser. This function takes a grammar, `g`, and a list of input tokens, `input`. The function returns a Boolean value, indicating whether `input` can be derived in the given grammar.

If we call this function using "`tryDerive g`", it will return a parser that can parse any correct input written using this LL(1) grammar. This is done by currying, which is by default in OCaml.

Example: See provided testcases.

3.14 Function tryDeriveTree

```
type parseTree =
| Terminal of string
| Nonterminal of string * parseTree list
```

```
tryDeriveTree (g : grammar) (inputStr : string list) : parseTree
```

This function is for **extra credit**. This function takes the same input as `tryDerive`. If `input` can be derived from the start symbol, then it returns a parse tree for `input`. We will only test this function on inputs that can be derived.

The parse tree is represented by an object of type `parseTree`. This type has two constructors. `Terminal sym` represents a single terminal symbol `sym`. `Nonterminal (sym, child_list)` represents a single non-terminal symbol `sym`, with child nodes listed in `child_list`. If `sym` expands into ϵ, then `child_list` is an empty list.

Example: See provided testcases.

4 Testing and Debugging

There are three ways to test your `proj2.ml` implementation.

Method 1: The first way is to work inside `ocaml`, the interactive OCaml interpreter. Since `proj2.ml` depends on `proj2_types.ml`, you will need to compile `proj2_types.ml` into an object file (which can be done by typing `make proj2_types.cmo`) first. You will need to type the following command in the interpreter to load it:

```
#load "proj2_types.cmo"
```

Next you can type `#use "proj2.ml"`. If no error occurs, you can then play with your functions and see their output. If you make any changes to your `proj2.ml` file you will have to type `#use "proj2.ml"` again. Alternatively, if you want to program in the interpreter environment, you do not have invoke `#use "proj2.ml"`. Later you can copy the implementation back to the `proj2.ml` after you have finished testing.

Method 2: The second way is to take advantage of the helper functions provided in the `proj2_driver.ml` file. Similarly, you will have to load the object files that `proj2_driver.ml` depends on. To do this, you need to type the following four lines in the shown order:

```
#load "str.cma"
#load "proj2_types.cmo"
#mod_use "proj2.ml"
#use "proj2_driver.ml"
```

If you have modified `proj2.ml`, you will have type to `#mod_use "proj2.ml"` again.

We provide a number of helper functions for you to use in `proj2_driver.ml`. The functions include but are not limited to

1. `printSymbolSet`: This function takes a symbol set and prints out its content.
2. `printSymbolMap`: This function takes a symbol map and prints out its content.
3. `printPredictSet`: This function takes a list of predict sets (as returned by `getPredictSets`) and prints out their content.
4. `printTree`: This function takes a parse tree (as returned by `tryDeriveTree` and prints the tree in a simple format.
5. `printTreeGraphviz`: Similar to `printTree`. However, the tree will be printed as a Graphviz script. You can convert Graphviz scripts into vector images here: http://www.webgraphviz.com/

Method 3: The third way is to use the file `codetest.ml`. If you invoke `make test` in the terminal, it will automatically compile all the object files that `codetest.ml` depends on (if they are not compiled or outdated) and run the code in `codtest.ml`. The results will be displayed in the terminal. The `codetest.ml` file contains example commands on how to test your `first, follow, predict` sets, as well as the generated parser.

The `codetest.ml` file also contains code on how to use the provided test cases. There are five test cases provided in the `testcases_simple/` folder. If you look at line 10 in `codetest.ml`, it contains this statement `#use "testcases_simple/testcase1.ml"`, which loads the `testcase1.ml` file for testing. You can change `testcase1.ml` to `testcase$i$.ml`, where $i$ is 1 to 5, representing one of the five test cases we provided. An initial run of the `codetest.ml` file (through `make test`) will yield something like "test failed", since all the function implementations are empty.

The `codetest.ml` file is a demonstration of how to make use of the functions you implemented in `proj2.ml`. You can also write code similar to that in `codetest.ml` for your own testing. You will not need to submit `codetest.ml`. This is not the final grading script. In the backend, we will have a rigorous auto-grader that checks every function you have implemented. However, the grammars in the five test cases will be covered by the auto-grader.

5 Grading

You will submit a tar file `proj2_$(NETID).tar.gz` that contains your modified `proj2.ml`. Please invoke `make submission` (on ilab) or `make NETID=[your netid] submission` (on your own machine) to generate this tar file. Your programs will be graded primarily on functionality.

You are given five test cases. We will use ten more secret test cases.

You should NOT use imperative features of OCaml for this project. For this reason, the following keywords of OCaml are forbidden. The grader includes a script to detect their usage.
`begin, class, do, done, downto, external, for, inherit, initializer, method, mutable, new, private, ref, to, virtual, while`

Also, you should NOT use any module from the standard library other than `Set` and `Map`.

Last but not least, this project is set up with respect to the configurations in the ilab machine. You can develop, run, and test your code on your computer. However, it is your responsibility to make sure your code will compile and run on ilab machines.

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.