Name: Jarrod Jeffrey Ingram SID: 430165568 Tutor: Mitchell

COMP2022 Assignment 2

Question 1

In the given grammar G there are two instances of FIRST/FIRST conflicts with a single instance of a special case of FIRST/FIRST conflicts, left recursion. Each of these three conflicts have simple solutions which are resolved below.

The first instance of a FIRST/FIRST conflict is in the production rules

```
L     → I ; L | I
```

In this case we have `FIRST(I ; L)` conflicting with `FIRST(I)`. This can be resolved by left factoring on `I` and introducing the new variable `L1`. The new form of this part of the grammar is

```
L     → I L1
L1    → ; L | ε
```

The second instance of a FIRST/FIRST conflict is the special case of left recursion. The particular production rules in question

```
E     → E Op1 E2 | E2
```

has `E` as the first part of `E → E Op1 E2` resulting in a FIRST/FIRST conflict with all alternatives, namely `E → E2`. The appropriate form for this part of the grammar introduces the variable `E1` providing the new form

```
E     → E2 E1
E1    → Op1 E2 E1 | ε
```

The final instance of a FIRST/FIRST conflict is similar in nature to the first. There's a conflict between `FIRST(T Op2 E2)` and `FIRST(T)`. Naturally this can be resolved in the same manner as the first conflict. Specifically, the new variable E3 is introduced resulting in

```
E2    → T E3
E3    → Op2 E2 | ε
```

These three modifications to `G` give the equivalent grammar `G'`:

```
P     → L
L     → I L1
L1    → ; L | ε
I     → A | C | W
A     → id := E
C     → if E then L O endif
O     → else L | ε
W     → while E do L end
E     → E2 E1
E1    → Op1 E2 E1 | ε
E2    → T E3
E3    → Op2 E2 | ε
T     → c | id
```

```
Op1   → < | = | !=
Op2   → + | -
```

Question 2

The following section details how the FIRST and necessary FOLLOW sets came about for the grammar `G′` and provides said FIRST and FOLLOW sets. These sets are then used to confirm whether or not the grammar `G′` is LL(1).

In the process of calculating the FIRST and necessary FOLLOW sets for all rules of `G′` the following algorithms were used:

FIRST Let $A \rightarrow X_1, \ldots, X_n$ be a production rule where X_i is either a terminal or variable. To compute FIRST($X_1, \ldots, X_n$), we first consider the leftmost symbol, X_1.

1. If X_1 is a terminal symbol, then FIRST($X_1, \ldots, X_n$) = X_1.
2. If X_1 is a variable, then FIRST($X_1, \ldots, X_n$) = X_1 contains the union of all FIRST sets for each right hand side of rules produced by X_1 without ε.
3. If X_1 can produce ε, then FIRST($X_1, \ldots, X_n$) depends on X_2. Repeat with X_2. Similarly, if both X_2 and X_2 can produce ε, we consider X_2, then X_2, etc.
4. If all $X_1, \ldots, X_n$ can generate ε then add {ε} to FIRST($X_1, \ldots, X_n$).

FOLLOW(A) If $X_1, \ldots, X_n \Rightarrow \varepsilon$ then consider the production rules where A appears in the *right hand side*. Let $B \rightarrow Y_1, \ldots, Y_n$ **A** $Z_1, \ldots, Z_n$ where Y_i, Z_i are either a terminal or variable.

1. FIRST($Z_1, \ldots, Z_n$) \ {ε} ⊂ FOLLOW(A).
2. If ε is in FIRST($Z_1, \ldots, Z_n$) then FOLLOW(B) ⊂ FOLLOW(A).

Given the above algorithms, it follows that the FIRST sets are

```
FIRST(P)    = FIRST(L) = FIRST(I)
            = FIRST(A) ∪ FIRST(C) ∪ FIRST(W)
            = {id, if, while}

FIRST(L1)   = FIRST(;) ∪ FIRST(ε)
            = {;, ε}

FIRST(O)    = FIRST(else) ∪ FIRST(ε)
            = {else, ε}

FIRST(E)    = FIRST(E2) = FIRST(T)
            = FIRST(c) ∪ FIRST(id)
            = {c, id}

FIRST(E1)   = FIRST(Op1) ∪ FIRST(ε)
            = (FIRST(<) ∪ FIRST(=) ∪ FIRST(!=)) ∪ FIRST(ε)
            = {<, =, !=, ε}

FIRST(E3)   = FIRST(Op2) ∪ FIRST(ε)
            = (FIRST(+) ∪ FIRST(-)) ∪ FIRST(ε)
            = {+, -, ε}
```

and that the necessary FOLLOW sets are

```
FOLLOW(L1) = FOLLOW(L)
           = FOLLOW(P) ∪ FOLLOW(L1) ∪ FIRST(O) \ {ε} ∪ FOLLOW(O)
           ∪ FIRST(end)
           = {$, else, endif, end}

FOLLOW(O)  = FIRST(endif)
           = {endif}

FOLLOW(E1) = FOLLOW(A) ∪ FIRST(then) ∪ FIRST(do)
           = FOLLOW(I) ∪ FIRST(then) ∪ FIRST(do)
           = (FIRST(L1) \ {ε} ∪ FOLLOW(L1)) ∪ FIRST(then)
           ∪ FIRST(do)
           = {;, $, else, endif, end, then, do}

FOLLOW(E3) = FOLLOW(E2)
           = FIRST(E1) \ {ε} ∪ FOLLOW(E1)
           = {<, =, !=, ;, $, else, endif, end, then, do}
```

Hence, the grammar is LL(1) as there are no FIRST/FIRST conflicts and no FIRST/FOLLOW conflicts. In the case of FIRST/FIRST conflicts every alternative but `I → A | C | W` is trivially disjoint, with

```
FIRST(A) ∪ FIRST(C) ∪ FIRST(W) = {id, if, while}
```

being a disjoint union as

```
FIRST(A) ∩ FIRST(C) ∩ FIRST(W) = ∅
```

It also happens that each necessary FOLLOW set is disjoint from the respective FIRST set which avoids FIRST/FOLLOW conflicts.

Question 3

As we have already computed the FIRST and FOLLOW sets from the previous question, most of the required work for constructing the parse table for a grammar has been completed. In most circumstances the rule that the parser will choose is based on the FIRST set of a particular variable. For example, in the case of the variable P we see that it resolves to L when one of the tokens in `{id, if, while}` is encountered.

However, the FIRST set of a variable is not always sufficient to determine which rule should be used next by the parser. The reason that we have the necessary FOLLOW sets from the previous question is that whenever a variable can resolve to ε it is popped from the stack if there's a token that can follow it.

As previously stated the algorithm for constructing the parse table once FIRST and FOLLOW sets have been computed is simple. Steps to fill the table T are as follows:

1. If there is a rule $R \rightarrow \alpha$ with $a \in$ FIRST(α), put α in T[R,a]
2. If there is a rule $R \rightarrow \alpha$ with $\varepsilon \in$ FIRST(α) and $a \in$ FOLLOW(R), put α in T[R,a]

From the previous question, it asked to confirm whether or not the grammar is LL(1) and we indirectly proved this using the absence of FIRST/FIRST and FIRST/FOLLOW conflicts as evidence. However the definition of an LL(1) parser dictates that a grammar is LL(1) only when there is at most one rule in every one of its cells in its parse table. Fortunately this is true for the particular grammar and the resulting parse table can be found below.

	;	id	:=	if	then	endif	else	while
P		P → L		P → L				P → L
L		L → I L1		L → I L1				L → I L1
L1	L1 → ; L					L1 → ε	L1 → ε	
I		I → A		I → C				I → W
A		A → id := E						
C				C → if E then L O endif				
O						O → ε	O → else L	
W								W → while E do L end
E		E → E2 E1						
E1	E1 → ε				E1 → ε	E1 → ε	E1 → ε	
E2		E2 → T E3						
E3	E3 → ε				E3 → ε	E3 → ε	E3 → ε	
T		T → id						
Op1								
Op2								

	do	end	c	<	=	!=	+	−	$
P									
L									
L1		L1 → ε							L1 → ε
I									
A									
C									
O									
W									
E			E → E2 E1						
E1	E1 → ε	E1 → ε		E1 → Op1 E2 E1	E1 → Op1 E2 E1	E1 → Op1 E2 E1			E1 → ε
E2			E2 → T E3						
E3	E3 → ε	E3 → ε		E3 → ε	E3 → ε	E3 → ε	E3 → Op2 E2	E3 → Op2 E2	E3 → ε
T			T → c						
Op1				Op1 → <	Op1 → =	Op1 → !=			
Op2							Op2 → +	Op2 → −	

Question 4

The parser for the parse table given above was successfully implemented. As per the first point in Question 4 the program outputs the leftmost derivation and the stack contents for each step. An example of this output using the test `accept-assign.txt`.

INPUT

```
id := c + id
```

OUTPUT

```
P      P $
L      L $
I L1   I L1 $
A L1   A L1 $
id := E L1       id := E L1 $
id := E L1       := E L1 $
id := E L1       E L1 $
id := E2 E1 L1   E2 E1 L1 $
id := T E3 E1 L1      T E3 E1 L1 $
id := c E3 E1 L1      c E3 E1 L1 $
id := c E3 E1 L1      E3 E1 L1 $
id := c Op2 E2 E1 L1   Op2 E2 E1 L1 $
id := c + E2 E1 L1     + E2 E1 L1 $
id := c + E2 E1 L1     E2 E1 L1 $
id := c + T E3 E1 L1   T E3 E1 L1 $
id := c + id E3 E1 L1       id E3 E1 L1 $
id := c + id E3 E1 L1       E3 E1 L1 $
id := c + id E1 L1     E1 L1 $
id := c + id L1   L1 $
id := c + id      $
accepted
```

The resulting output from the test shows that the leftmost derivation is first, with a tab for separation and then the stack contents. It is also illustrates that the parser finally prints out accepted or rejected before exiting to fulfil the second point of Question 4.

To realise the third and fourth points of Question 4 the test `reject-empty-while-condition-body.txt` will be used.

INPUT

```
while do
end
```

OUTPUT

```
P      P $
L      L $
I L1   I L1 $
W L1   W L1 $
while E do L end L1    while E do L end L1 $
while E do L end L1    E do L end L1 $
expected 'c' or 'id' instead of 'do'
while do L end L1      do L end L1 $
while do L end L1      L end L1 $
expected 'id', 'if' or 'while' instead of 'end'
while do end     L end L1 $
rejected
```

In the given example we can see that the third point of Question 4 has been realised. The parser was expecting a constant or identifier for the required expression in the condition part of a `while` statement. The parser instead finds a `do` terminal and subsequently prints an appropriate error message. In general the error reporting uses the FIRST set of the expected variable or terminal. However in the case of variables that can produce ε the parser can be expecting a wide range of tokens so the FOLLOW set of any of these variables is also included in the error report. In this output the parser has successfully recovered from the first error and continues parsing, which leads us into the implementation of the fourth point of Question 4; error recovery.

The parser performs error recovery by using a set of "synchronising" tokens for a particular variable. The set of synchronising tokens is the union of the set of terminals found in the FIRST and FOLLOW sets for a variable as well as a set of START terminals; terminals which can start statements. In the given grammar the set of START terminals are those in `FIRST(I) = {id, if, while}`. In this test it happens that we've failed on `E` with the resulting set of synchronising tokens in

```
SYNCH(E) = {c, id, do, then, ;, end, else, endif, $, if, while}
```

Given the input the parser is able to synchronise on the `do` token and continue parsing. The parser then encounters the second error of an empty body and prints an appropriate error message. After reporting this error the parser is able to recover on the `end` token as

```
SYNCH(L) = {id, if, while, end, else, endif, $}
```

but given the stack the parser is forced to bail out of the parsing and reject the input.

The algorithm in use for recovering from an error is the following:

1. If the top of the stack is a top-level rule then stop parsing; top-level rules have a knack for creating spurious error messages. Top-level rules are those that can occur before any terminals have entered the derivation.
2. If there is no more input to synchronise with then stop parsing.
3. If the current token is in the FOLLOW set of the top of the stack then continue parsing.
4. If the current token is in the START set then keep popping items off the stack until we find an item that has one of the terminals in the START set in its FIRST set. Once found continue parsing. If we don't have anything left on the stack then stop parsing.
5. If the current token is in the FIRST set of the top of the stack then push the top of the stack back onto the stack and continue parsing.
6. Otherwise go to the next token and repeat from 2.

This method of error recovery is known as panic mode error recovery. It's called such because when the parser encounters an error it starts panicking by skipping input until an appropriate synchronising token can be found to continue parsing with. As with all error recovery there's caveats to this method. The biggest caveat being that large portions of input can be skipped resulting in various errors being missed. However panic mode error recovery is a simple technique to implement compared to some of the other methods and provides effective results. Other methods include the insertion and deletion of tokens to repair the input or ad-hoc methods; ad-hoc methods are particularly time consuming to implement but most effective in their results.[1]

[1] Grune, D & Jacobs, JHC 2008, *Parsing Techniques: A Practical Guide*, 2nd edn, Springer Science+Business Media LLC, New York, NY

Question 5

Adding or modifying a rule in any grammar can have profound implications on the other rules in the grammar. Therefore it's assumed that we have recomputed the FIRST and FOLLOW sets and corrected the parsing table for the entire grammar given whatever modifications or additions were made.

The changes required by the program to parse the new grammar are minor given that the program is data-driven. By adding or modifying entries in the dictionary associating variables with rules, the program will now be able to parse the new grammar but won't be able to perform correct error reporting.

To perform correct error reporting the dictionaries associating variables with their FIRST sets and also the table associating variables with their FOLLOW sets would have to be modified to accommodate the changes in the recomputed FIRST and FOLLOW sets. If there is any modifications or additions to the terminals that can start statements then the list of start terminals would have to be changed. Finally if there are any modifications or additions to the top-level rules they will need to be added to the respective list.

Question 6

The testing framework employed is particularly simple and therefore effective to use. The information required for the expected output and a description of the test is contained in the filename for each test. Filenames of the form `accept−*` are expected to return "accepted" and filenames of the form `reject−*` are expected to return "rejected". The rest of the filename is used as a description of what the test does.

The set of tests covers a wide-range of cases that the grammar is expected to handle. The accept cases revolved around different levels of complexities in terms of combinations of various constructs in the language. Some examples being two statements in a row or a loop nested within a conditional statement.

The set of reject cases revolved around missing constructs in the language such as an empty loop body, invalid tokens such as "the" instead of "then" and missing terminals such as no "end" to a loop. The latter also had varying levels of complexity tested like missing the "end" in a loop which is nested within a conditional.

The test suite uses all files in the `t/` directory to perform the tests. Each test has output of the form

```
description: {passed, failed}: expected {accepted, rejected},
observed {accepted, rejected}
```

with the final line of output being a summary of the test suite

```
[number of tests passed]/[total number of tests] tests passed
```