COMP2022: Assignment 2

Due: 23:00pm Sunday 14th October 2018 (end week 10)

1 The grammar G [10%]

Consider the following grammar G, which represents a fragment of a simple programming language:

$$\begin{aligned}
&L \to LE \mid E\\
&E \to (C) \mid (F) \mid V \mid T\\
&C \to \texttt{if}EE \mid \texttt{if}EEE\\
&F \to +L \mid -L \mid *L \mid \texttt{print}L\\
&V \to a \mid b \mid c \mid d\\
&T \to 0 \mid 1 \mid 2 \mid 3
\end{aligned}$$

i) List the variables of G

ii) List the terminals of G

iii) What is the start variable of G?

iv) Give a leftmost derivation of the string $(\texttt{if}(-1a)(\texttt{print}1))$

v) Draw a parse tree for $(\texttt{if}(-1a)(\texttt{print}1))$

2 Prove G is not LL(1) [10%]

Prove that G is not an LL(1) grammar.

3 Find an equivalent LL(1) grammar G' [10%]

Find an equivalent grammar G' which is LL(1), by using the grammar transformation techniques shown in lectures, or otherwise. Describe the process and show your working.

4 LL(1) parse table [15%]

Complete the LL(1) parse table for G'. Describe the process and show your working, including:

1. FIRST sets for all the *production rules* of G'
2. FOLLOW sets for variables of G' *only if they are needed*

5 Implementation [25%]

Implement a program which parses strings using an **LL(1) table driven parser**, using the table you determined for G' in the previous exercise. You may use Python, Java, C, C++, or Lisp. If you'd like to use a different language then please check with us first.

- Input:
  The first *command line argument* is the filename of a file containing the string of characters to test.
- Output:
  1. Print a trace of the execution, showing the steps followed by the program as it performs the left-most derivation. This should look similar to parsing the string through a PDA. An example of this is given in the appendices.
  2. After parsing the whole input file, print `ACCEPTED` or `REJECTED`, depending on whether or not the string could be derived by the grammar.
  3. If there is a symbol in the input string which is not a terminal from the grammar, the program should output `ERROR_INVALID_SYMBOL` (This could be during or before trying to parse the input.)
- All whitespace in the input file should be ignored (line breaks, spaces, etc.). The output will be easier to read if you remove the whitespace *before* starting the parse.
- Examples of the program output syntax are provided in the appendices.

6 Extension [30%]

You may pick one of the following extensions to improve your parser. Any *one* of the following ideas would be sufficient (do not implement more than one). They are listed roughly in order of increasing difficulty/effort, but are worth the same marks:

a) Use the FIRST and FOLLOW sets to implement an error recovery feature. This should give the user suggestions on possible corrections which could change strings which could not be derived in G' For this extension you need to:

   - Implementation: If a second command line argument `error` is given, then instead of rejecting a string that is not in the language, it should make suggestions about how to correct it. The user chooses one of the options, then the program should continue the parse. Some examples are provided in the appendices. This should be included in the code you submit to PASTA.
   - Report: Explain how your error recovery feature uses the FIRST and FOLLOW sets to work, and show some useful examples.

b) Extend your program to be able to evaluate the input program, if a second command line argument `eval` is given. Some examples are provided in the appendices, and some supplementary material will be provided on Ed to explain the algorithm needed to build and evaluate the parse tree. This should be included in the code you submit to PASTA.

   - Implementation:
     – Build a parse tree as it performs the leftmost derivation.
     – Evaluate that parse tree.
   - Report: Give some examples of usage

c) Implement a second parsing algorithm. You could implement:

   - the CYK algorithm, or

- a LR(1) parser
- Note: a recursive descent parser would *not* be sufficient (too easy).

You will need to submit your code to PASTA, as well as including some details in your report comparing your second parser to the first one (what are the advantages and disadvantages of this parser compared to the LL(1) table driven parser.)

d) Something else? Check with your tutor to see if they think it's appropriate.

7 Submission details

Due 23:00pm Sunday 14th October 2018. The late submission policy is detailed in the administrivia lecture slides from week 1. Please notify us if you intend to make a late submission, or if you believe you will not be able to submit, to make it easier for us to support you.

7.1 Canvas submission

You must submit a report as a single document (.pdf or .docx) to TurnItIn via Canvas. The written parts of the report must be *text*, not images of hand-writing. Any diagrams can be images, of course. The report should include:

- Task 1, 2, 3, 4: Your answers, working, and explanations
- Task 5: A description of the testing runs you used, including examples of the output. Show enough to convince the marker that your testing was comprehensive
- Task 6: A description of any additional work you did, including documentation about how to use it and some examples of input/output

7.2 PASTA submission

Exact submission details (such as requirements on the name of the program) will be provided next week.

- Task 5: submit your source code.
- Task 6: submit your source code, if relevant

The *visible* tests for task 5 (the implementation) will be just enough to show you that you have got the input/output syntax correct. It will not test the correctness of your program (you are expected to test that yourself!)

There will be no automatic testing of any extra functionality you added as an extension.

7.3 Marking criteria

The weight of marks for each question are noted next to each question.

- Tasks 1, 2, 3, 4: The marks will be roughly evenly divided between correctness, and on your working and explanations.
- Task 5: The marks will be roughly evenly divided between automatic marking (for correctness) and hand marking (based on your testing, the quality of your explanations, code, etc.)
- Task 6: The extension will be hand marked.

8 Appendices

8.1 Example of string derivations

Suppose we had a program which parsed this grammar fragment:

$$
\begin{aligned}
E &\to (F) \mid T && \text{(start variable)}\\
F &\to +EE\\
T &\to 0 \mid 1 \mid 2 \mid 3
\end{aligned}
$$

Example 1: For the input:

```
(+12)
```

The output might look like this (note: it doesn't need to line up neatly):

```
(+12)$        E$
(+12)$      (F)$
 +12)$       F)$
 +12)$     +EE)$
  12)$      EE)$
  12)$      VE)$
  12)$      1E)$
   2)$       E)$
   2)$       V)$
   2)$       2)$
    )$        )$
     $         $
ACCEPTED
```

Example 2: For the input:

```
     (   +
1
    3     )
```

The output is the same, because we ignore all the whitespace:

```
(+12)$        E$
(+12)$      (F)$
...
ACCEPTED
```

Example 3: For the input:

```
(++3)
```

We get:

```
(++3)$        E$
(++3)$      (F)$
 ++3)$       F)$
 ++3)$     +EE)$
  +3)$      EE)$
REJECTED
```

It is rejected because there would be no entry in the parse table for $(E, +)$.

8.2 Extension option A (error recovery)

Reminder: your code should only run the extension if a second command line argument `error` is given. Otherwise it should behave normally.

This appendix shows a few examples of what using an error recovery feature might look like. Your output and features do not need to be identical to these examples, it is only a guideline to give you some ideas.

Suppose we had a program which parsed this grammar fragment (your grammar will be different!):

$$
\begin{aligned}
E &\to (F) \mid T && \text{(start variable)}\\
F &\to +EE\\
T &\to 0 \mid 1 \mid 2 \mid 3
\end{aligned}
$$

There are three key steps to implementing error recovery. More marks will be awarded to more useful/sophisticated features.

8.2.1 Identifying the error

The first step is to be able to describe the error. Examples:

```
(+1+)$       E$
(+1+)$     (F)$
 +1+)$      F)$
 +1+)$    +EE)$
  1+)$     EE)$
  1+)$     VE)$
  1+)$     1E)$
   +)$      E)$
Error: got +, but expected {0, 1, 2, 3, (}.
REJECTED
```

```
    1)$       E$
    1)$       T$
    1)$       1$
     )$        $
Error: got ), but expected $.
REJECTED
```

8.2.2 Recovering with user intervention

The next step would be to give the user the option to correct the error and allow the derivation to continue.

For example, you might have an option to delete the next input symbol:

```
(+123)$       E$
(+123)$     (F)$
 +123)$      F)$
 +123)$    +EE)$
  123)$     EE)$
  123)$     VE)$
  123)$     1E)$
   23)$      E)$
   23)$      V)$
   23)$      2)$
```

```
    3)$       )$
Error: got 3, but expected {)}.
Delete input? Y
     )$       )$
      $        $
ACCEPTED (+12)
```

It's a bit more powerful if we also allow the user to insert a symbol:

```
(+12$       E$
(+12$     (F)$
 +12$      F)$
 +12$    +EE)$
  12$     EE)$
  12$     VE)$
  12$     1E)$
   2$      E)$
   2$      V)$
   2$      2)$
    $       )$
Error: got $, but expected {)}.
Add input? )
   )$       )$
    $        $
ACCEPTED (+12)
```

Better still, would be to allow the user to choose to delete or insert (so they can replace a symbol)

8.2.3 Making useful suggestions

You might automatically suggest which is the 'best' correction. For example, you might have the program explore each option automatically, then recommend the correction which leads to an accepted string using a minimal number of changes. For performance reasons, it would be wise to limit the maximum number of changes to some fairly small number (e.g. no more than 5 changes).

8.3 Extension option B (evaluation)

Reminder: your code should only run the extension if a second command line argument `eval` is given. Otherwise it should behave normally.

8.3.1 Semantics of the language:

- **(+ L)** := the sum of the expressions.
- **(- L)** := subtract all the remaining expressions from the first one. If there is only one expression, take the negation of it.
- **(* L)** := the product of the expressions. If there is only one expression, return it.
- **(if E E)** := if the first E is not zero, return the second one. Otherwise, return zero.
- **(if E E E)** := if the first E is not zero, return the second one, otherwise return the third one.
- **(print L)** := print (the result of) each of the expressions, on separate lines, and return the value of one of them. (The order and which is returned is not defined, you may decide this).
- If there is more than one expression in the program, they should be evaluated in order. The only output is the print statements!

8.3.2 Examples of output:

Input	Output	Why
`(print (+ 3))`	3	
`(print (+ 3 2))`	5	$(3 + 2) = 5$
`(print (+ 3 2 1))`	6	$(3 + 2 + 1) = 6$
`(print (- 3))`	-3	Negation
`(print (- 3 2))`	1	$(3 - 2) = 1$
`(print (- 3 2 1))`	0	$(3 - 2 - 1) = 0$
`(print (* 4))`	4	
`(print (* 4 3))`	12	$(4 * 3) = 12$
`(print (* 4 3 2))`	24	$(4 * 3 * 2) = 24$
`(print 1)(print 2)`	1 2	The statements are evaluated in order
`(print (+ 1 (print 2))`	2 3	The nested print expression must be evaluated first, and returns 2 to the addition
`(if 0 (print 2))`		no print statement is evaluated
`(if 1 (print 2))`	2	
`(print (if 1 2))`	2	
`(print (if 0 2))`	0	The if statement was false, and had no else block, so it evaluated as 0
`(print (if 0 2 1))`	1	

8.3.3 More information:

There is an announcement on Ed `https://edstem.org/courses/2892/discussion/108704`
It contains:

- Slides explaining the concepts
- An example of the process of building a parse tree using this algorithm
- Some code examples of evaluating a parse tree for a simpler grammar (in Python and Java)