COMP2022: Formal Languages and Logic
Assignment 2

Due: 23:59pm Thursday 18th May 2017 (week 10)

Consider the following grammar G:

$P \to PL \mid L$
$L \to N\texttt{;} \mid M\texttt{;} \mid C$
$N \to \texttt{print}\ E$
$M \to \texttt{print "}W\texttt{"}$
$W \to TW \mid \varepsilon$
$C \to \texttt{if}\ E\ \{P\} \mid \texttt{if}\ E\ \{P\}\ \texttt{else}\ \{P\}$
$E \to \texttt{(}EOE\texttt{)} \mid V$ (note: this has a variable O)
$O \to \texttt{+} \mid \texttt{-} \mid \texttt{*}$
$V \to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3}$ (note: this has a terminal `0` (zero))
$T \to \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d}$

1 Define G [10%]

- Show the set of variables of G
- Show the set of terminals of G
- What is the start variable of G?

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

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

3 Transforming G [15%]

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

5.1 Parsing strings with an LL(1) table driven parser [20%]

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 Haskell. 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.)

  Note: 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.

5.2 Evaluating programs written in G' [20%]

If a *second* command line argument "eval" is given, then *instead* of printing the trace of the parse, your program should:

1. Build a parse tree as it performs the leftmost derivation (see the week 7 lecture and tutorials.)
2. Evaluate that parse tree (see the week 7 lecture and tutorials.)

The semantics (meaning) which we are applying to our rules are as follows:

- V variables derive integers
- W variables derive strings
- E expressions are evaluated like normal integer arithmetic
- `print E` statements *output* (to screen) the result of evaluating the expression E. (i.e. `print (1+1)` outputs 2)
- `print "W"` statements *output* the string derived from W (i.e. `print "abba"` outputs abba)
- `if` statements evaluate the contents of their *if* block if and only if the *condition* evaluated to a non-zero value, otherwise the *else* block is evaluated instead (if there is one).

If the input could not be parsed then output `REJECTED` instead.
Some examples of programs and their expected output are provided in the appendices.

6 Extension [10%]

This can be an extension to your program, a complementary tool, or perhaps some extra written work. Any *one* of the following ideas would be sufficient:

- Discuss in your report how you might use FIRST and FOLLOW sets to implement some sort of error recovery feature. i.e. to give the user suggestions on possible corrections which could change strings which could not be derived in G'. You don't need to implement it, but you should at least provide some pseudocode and examples of expected output.
- Compare and discuss the merits of at least 3 parsing algorithms (without implementation) (For example, you might compare the LL(1) table driven parser we used, the CYK algorithm, and an LR parser.) You don't need to implement them.
- Extend the grammar to support variable assignment, and add this to your implementation
- Extend the grammar and your implementation to support the ternary if operator in expressions - $E \rightarrow (E?E : E)$ where $(x?y : z)$ evaluates as y if x is non-zero, otherwise z. Note: you will need to transform the grammar to LL(1) again, and evaluation of the parse tree will be significantly more difficult to implement.
- Implement an alternative parsing algorithm, such as the CYK algorithm.
- Something else? Check with your tutor to see if they think it's appropriate.

Note that the marks allocated to the extension are *not* proportional to the work involved. Some of the suggestions are substantially more challenging than others. You should complete the rest of the assignment *before* considering an extension.

7 Submission details

Due 23:59pm Thursday 18th May 2017. The late submission policy is detailed in the administrivia lecture slides from week 1. Please notify me if you intend to make a late submission, or if you believe you will not be able to submit, to make it easier for me to support you.

7.1 Cover sheet submission

You must submit a signed cover sheet to eLearning:

`http://sydney.edu.au/engineering/it/current_students/undergrad/guidelines/assignment_sheet_individual.pdf`

7.2 TurnItIn (eLearning) submission

You must submit a report as a single document (.pdf or .docx) to TurnItIn. 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 5.1: 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 5.2: A description of the testing runs you used, including examples of the output. Show enough to convince the marker that your testing was comprehensive
- Extension: A description of any additional work you did, including documentation about how to use it and some examples of input/output

7.3 Ed submission

- Task 5.1, 5.2: submit your source code. You must edit the "build.sh" and "run.sh" files in Ed to match the language you are using (and the file/class names you used.) These files tell Ed how to compile and run your program. By default they expect your program to be called "parser" (or "Parser" for Java programs), but you can alter this by editing these scripts. You can also add any compiler flags etc. that you need.

- Extension: 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.

7.4 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.

- Tasks 5.1, 5.2: 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 entirely hand marked.

8 Appendices

8.1 Example of string derivations

Suppose we had a program which parsed this grammar fragment:

$$E \to (EOE) \mid V$$
$$O \to + \mid - \mid *$$
$$V \to 0 \mid 1 \mid 2 \mid 3$$

Example 1:
For this input:

```
(1+0)
```

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

```
(1+0)$        E$
(1+0)$    (EOE)$
 1+0)$     EOE)$
 1+0)$     VOE)$
 1+0)$     1OE)$
  +0)$      OE)$
  +0)$      +E)$
   0)$       E)$
   0)$       V)$
   0)$       0)$
    )$        )$
     $         $
ACCEPTED
```

Example 2:
For this input:

```
     (   1
+
   0 )
```

The output is the same, because we ignore whitespace:

```
(1+0)$        E$
(1+0)$    (EOE)$
 1+0)$     EOE)$
 1+0)$     VOE)$
 1+0)$     1OE)$
  +0)$      OE)$
  +0)$      +E)$
   0)$       E)$
   0)$       V)$
   0)$       0)$
    )$        )$
     $         $
ACCEPTED
```

8.2 Examples of program evaluation

Input	Output	Why
`print (2+2);`	4	
`print "b"; print "abba";`	b abba	
`if 1 {print 1;} else {print 2;}`	1	The if block activated
`if 0 {print 1;} else {print "b";}`	b	The else block activated
`if (1-1) {print 1;} else {print 2;}`	2	(1-1) = 0, so the condition was false
`if (1+1) {print 1;} else {print 2;}`	1	(1+1) = 2, so the condition was true
`print 1; if 1 {print 2;} print 3;`	1 2 3	
`print 1; if 0 {print 2;} print 3;`	1 3	The if block didn't activate, but the following print still does
`if 1 {print 1;} if 1 {print 1;}`	1 1	
`print 1`	REJECTED	Invalid program (missing ;)
`print aa;`	REJECTED	Invalid program (missing ")