CSC4180 Assignment 3: Parsing Techniques

Release Date: March 18, 2025
Due Date: April 7, 2025
TA in Charge: Mr.LiuYuxuan

Any kind of plagiarism is not tolerated and will get zero point!!!

Q1: Resolve Ambiguity for Micro Language's Grammar (20%)

Recall the context-free grammar of Micro Language that we worked on in Assignment 1, you may now notice that the grammar turns out to be ambiguous, which means there exists multiple different parse trees for the same input string. **Please answer the following questions:**

1. An tricky way to argue that a grammar is ambiguous is to give a counter-example input that can be parsed in at least two different ways. Can you come up with such an example for Micro's grammar to show that the grammar is ambiguous? (10%)

2. However, to determine whether a grammar is not ambiguous can be hard, but we can take advantage of the LL(1) parsing table.

   (a) If there are multiply defined entries in the LL(1) parsing table of the grammar, can we say this grammar must be ambiguous? Answer "Yes" or "No" and briefly explain why within 50 words. (5%)

   (b) If there is no multiply defined entry in the LL(1) parsing table of the grammar, can we say this grammar is definitedly not ambiguous? Answer "Yes" or "No" and briefly explain why within 50 words. (5%)

Recall: Micro's Context-Free Grammar

Here is the extended CFG defining Micro Language:

1. <program> → BEGIN <statement list> END
2. <statement list> → <statement> {<statement>}
3. <statement> → ID ASSIGNOP <expression>;
4. <statement> → READ LPAREN <id list> RPAREN;
5. <statement> → WRITE LPAREN<expr list> RPAREN;
6. <id list > → ID {COMMA ID}
7. <expr list > → <expression> {COMMA <expression>}
8. <expression> → <primary> {<add op> <primary>}
9. <primary> → LPAREN <expression> RPAREN
10. <primary> → ID
11. <primary> → INTLITERAL
12. <add op> → PLUOP
13. <add op> → MINUSOP
14. <system goal> → <program> SCANEOF

Contents inside {} means that it can appear 0, 1 or multiple times.

Q2: Simple LL(1) and LR(0) Parsing Exercises (30%)

In this section, you are required to play with the web demos of LL(1) and LR(0) provided by Princeton University, and do some exercises related to these two parsing techniques.

LL(1) Parser Visualization Web Demo

LR(0) Parser Visualization Web Demo

Please answer every indexed sub-question in Q2 following the instruction below:

LL(1) Grammar (15%)

Here is a simple grammar and a corresponding input string to parse.

```
E −> T | T + E
T −> int | int * T | ( E )
```

```
Input String: int + int * int
```

1. Modify the grammar to make it LL(1)

   - You can use the LL(1) web demo to see which entries in the parsing table are multiply defined.
   - Recall your answer in Q1, find out what the problem is to the grammar and how to fix it. **(Hints: left recursion elimination or left factoring?)**

2. Translate your LL(1) grammar in the format that the LL(1) web demo can recognize, and then generate all intermediate tables, and parse the input string.

   - Notice that ϵ(empty string) should be denoted as two consequent single-quotes instead of one double-quotes.
   - Take a screenshot of the two tables (one is nullable, first and follow sets, and the other is parsing table) and include them in your report.

3. Perform the parsing step by step in the playground to get the final parse tree.

   - Take a screenshot of the parse tree generated in the web demo, and include it in your report.

LR(0) Grammar (15%)

1. Play with the LR(0) web demo with its default grammar

   - Take a screenshot of the LR(0) automaton (both table and DFA diagram), and include it in your report.

2. Try in web demos and see if the default LR(0) grammar also LL(1), if the sample LL(1) grammar also LR(0).

   - Briefly explain why the LL(1) grammar is not LR(0) with the help of the LR parsing table. You need to clearly specify which conflict occurs in LR(0) parsing and why that conflict fails the LR parsing instead of telling me that there is a red row in the parsing table.

3. It is well believed that *"It is difficult to write a bottom-up parser by hand for anything but trivial grammars."(Courtesy to Maggie Johnson of Stanford CS143's Bottom-Up Parsing Materials, 2012).* Explain your understanding to this argument based on your findings in the LR0 playground or other supporting materials.

Q3: Implement LL(1) Parser by hand for Oat v.1 (50%)

As a top-down parsing algorithm, LL(1) parsing is very intuitive and is easy to implement by hand. In this section, you are required to implement a top-down LL(1) parser by hand with the LL(1) grammar of Oat v.1 Language provided. No template will be provided and you can write your programs in anyway you like.

Important Checkpoints for Grading

1. Read the provided LL(1) grammar file of Oat v.1. Explain why the original production rules cannot be used directly for LL(1) parsing (in other words, not LL(1)). Just pick one production rule as example (like what you did in Q1), find its corresponding rule(s) in the provided LL(1) grammar file and see how an LL(1) grammar is constructed. (5%)

2. Parse the LL(1) grammar file as input, extract each production rule for further construction of parsing table (5%)

3. Compute the nullable and first set correctly (10%)

4. Compute the follow set correctly (10%)

5. Construct the LL(1) parsing table correctly (10%)

6. Parse input token streams correctly (10%)

7. A very brief report that covers how to compile and execute your program to get expected output.

Important Suggestions

- **Generating a concrete parse tree is enough. Abstract syntax tree (AST) is only for bonus.**

- The provided LL(1) grammar is well formatted for the web demo. Feel free to try the grammar online and verify all the intermediate results you generated with the ones shown in the web demo to make sure that you are on the right track.

- Constructing the LL(1) grammar for Oat v.1 Language is pretty hard. It has been tested on some of the testcases in Assignment 2, but it may fails to some other Oat v.1 programs. If you find any errors, please let the TA know and he will change the grammar.

Review of LL(1) Parsing Algorithm

1. Compute First and Follow Set

Refer to Figure 1: Algorithm to Compute Nullable, First and Follow Set

2. Construct the Parsing Table

Once we have the first and follow sets, we build a table M with the leftmost column labeled with all the nonterminals in the grammar, and the top row labeled with all the terminals in the grammar, along with \$. The following algorithm fills in the table cells:

1. For each production rule $A \rightarrow u$ of the grammar, do steps 2 and 3

2. For each terminal a in First(u), add $A \rightarrow u$ to M[A,a]

3. If ϵ in First(u), (i.e. A is nullable) add $A \rightarrow u$ to M[A,b] for each terminal b in Follow(A), If ϵ in First(u), and \$ is in Follow(A), add $A \rightarrow u$ to M[A,\$]

4. All undefined entries are errors

Initialize FIRST and FOLLOW to all empty sets, and nullable to all false.
for each terminal symbol Z
    $\text{FIRST}[Z] \leftarrow \{Z\}$
repeat
    for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
        for each i from 1 to k, each j from $i+1$ to k,
            if all the Y_i are nullable
                then nullable$[X] \leftarrow$ true
            if $Y_1 \cdots Y_{i-1}$ are all nullable
                then $\text{FIRST}[X] \leftarrow \text{FIRST}[X] \cup \text{FIRST}[Y_i]$
            if $Y_{i+1} \cdots Y_k$ are all nullable
                then $\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FOLLOW}[X]$
            if $Y_{i+1} \cdots Y_{j-1}$ are all nullable
                then $\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FIRST}[Y_j]$
until FIRST, FOLLOW, and nullable did not change in this iteration.

Figure 1: Algorithm to Compute Nullable, First and Follow Set

The concept used here is to consider a production $A \rightarrow u$ with a in First(u). The parser should expand A to u when the current input symbol is a. It is a little trickier when $u = \epsilon$ or $u \rightarrow *\epsilon$. In this case, we should expand A to u if the current input symbol is in Follow(A), or if the \$ at the end of the input has been reached, and \$ is in Follow(A). If the procedure ever tries to fill in an entry of the table that already has a nonerror entry, the procedure fails— the grammar is not LL(1).

3. The LL(1) Parsing Algorithm

Suppose a grammar has start symbol S and LL(1) parsing table T. We want to parse string ω in the following way:

1. Initialize a stack containing $S\omega$

2. Repeat until the stack is empty:

    - Let the next character of ω be t.
    - If the top of the stack is a terminal symbol r:
        - If r and t do not match, report an error.
        - Otherwise, consume the character t and pop r from the stack.
    - Otherwise, the top of the stack is a non-terminal symbol A:
        - If T[A, t] is undefined, report an error.
        - Replace the top of the stack with T[A, t].

Oat v.1 Language Specification

- Oat v.1 Language Specification Document (Reserved Words Highlighted)

- Oat v.1 Detailed Explanation by Prof.Steve Zdancewic, COS320: Compiling Techniques, University of Princeton

LL(1) Grammar for Oat v.1 Language (in LL(1) web demo format)

The txt file of the following LL(1) grammar for Oat v.1 Language has been uploaded to BlackBoard and our GitHub repository as well.

Oat v1 LL(1) Grammar

Bonus: (Extra Credits 10%)

If some of you want to challenge yourself for something harder or advanced, here are some alternatives that you can refer to:

1. Implement a Bottom-Up Parser using Bison (LALR) parser generator for Oat v.1 Language

- Take advantage of the flex scanner you implemented in Assignment 2.

- Be careful when dealing with the left associativity and precedence for binary operators in bison.

- You can choose parser generators other than bison to finish this task.

- Other extra work that you have done in Q3: handwritten LL(1) parser will also receive some extra credits.

2. Implement a Parser using ANTLR for Oat v.1 Language

ANTLR (ANother Tool for Language Recognition) is also a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build parse trees and also generates a listener interface (or visitor) that makes it easy to respond to the recognition of phrases of interest.

ANTLR4's GitHub Repository: https://github.com/antlr/antlr4

ANTLR's Tutorial: https://tomassetti.me/antlr-mega-tutorial/

3. Generate Abstract Syntax Tree from the Parse Tree

Obviously, there are too many redundant information and nodes in the generated parse tree, and you may remove those redundancy and get a clean version of AST.

Other extra work that you have done in Q3: handwritten LL(1) parser will also receive some extra credits.

Submission

Policy of Late Submission

- Late submission within 10 minutes after then DDL is tolerated for possible network issues during submission.

- 10 Points deducted for each day after the DDL (11 minutes late will be considered as one day, so be careful)

- Zero point if you submitted your project late for more than two days.

- Bonus points cannot make up for points lost due to late submission.

Submission with Source Code

```
csc4180−a3−118010200.zip
|−
|−−−− csc4180−a3−118010200−report.pdf
|−
|−−−− testcases
|−
|−−−− src
              | All  your  source  files
```

Submission with Docker

If you want to submit your program in a docker, your submission should look like:

```
csc4180−a3−118010200.zip
|−
|−−−− csc4180−a3−118010200.Dockerfile
|−
|−−−− csc4180−a3−118010200−report.pdf
|−
|−−−− src
         |−
         |−−−−Makefile
         |−
         |−−−−run_compiler.sh
         |−
         |−−−−Other possible files
```