Programming Language Concepts Lab 3 Instructions

Y2024 – 20/03/2024, Matthieu DE MARI

Introduction

This PDF file contains the instructions for the Lab 3 of the 50.051 Programming Language Concepts course at the Singapore University of Technology and Design.

It consists of four tasks, which build up progressively. Each task has questions leading to the expected solution, which will have to be coded sometimes.

Please submit your assignment as a Zip file. This zip may contain any number of .c files and .h files. For every .c file, there should be a .h file. Keep every file used from the lab template as they will be used for testing your code and as input to your program.

Your assignment does NOT have to compile using all the the flags -ansi -pedantic -Wall -Werror. For this Lab and the next one, we will simply relax this constraint, but good programming practices should still apply to your solutions.

Deadline: Thursday 7th April 2024, 11.59pm.

Submission should consist of: Your code files (TaskX.c and .h, if any), and a small PDF report, answering questions for each of the four tasks. Will have to be submitted on eDimension.

Task 1: Introduction to Context-Free Grammars

As we have seen in class, a Context-Free Grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings. A CFG consists of a set of **terminals** (symbols that appear in the strings), a set of **non-terminals** (symbols that help define the structure but do not appear in the strings), a set of **production rules** connecting symbols to other (combinations of) symbols, and a **start symbol** for the CFG.

In this Lab 3, we will first consider the following CFG.

(1): $S \rightarrow E$

(2): $E \rightarrow E + T$

(3): $E \rightarrow T$

(4): $T \rightarrow n \in \mathbb{N}$ (any positive integer number)

In the CFG above, we can identify the following components:

- **Terminals:** The basic symbols from which strings are formed. In the example grammar to be explored in this Lab 2, the terminal symbols will be positive integer numbers (denoted by the symbol n) and the + operator, which stands for the mathematical addition.
- **Non-terminals:** Symbols that can be replaced with sequences of terminals and non-terminals. In the example, S, E, and T are non-terminals.
- **Rules/Productions:** Rules that describe how to form strings from the language. Each rule consists of a non-terminal followed by an arrow (→) and a sequence of terminals and/or non-terminals. Our CFG consists of four production rules, denoted (1-4) and shown above.
- **Start Symbol:** A special non-terminal symbol from which the production starts. In the example, S is the start symbol for our CFG.

Question 1-A: What is the purpose of this CFG? What does it allow to check? How can this be used in our compiler?

Question 1-B: The production rule (2) is recursive. What does this mean that a production rule is recursive? What is the purpose of using such a recursion in our CFG?

This concludes Task 1.

Task 2: Coding a simple CFG

Description

In this section, we will discuss the implementation of a simple CFG framework, that we could use for the CFG introduced in Section 1.

You may open the code in the *CFG_basics.c* file. As you will see it starts with includes and defines, more specifically we have used three define statements to produce the following identifiers with the macro values:

- MAX_RHS: The maximal number of symbols on the RHS of a CFG production rule (arbitrarily set to 10, the largest number of symbols used is 3, in production rule $E \rightarrow E + T$).
- MAX_SYMBOLS: The maximal number of symbols in a CFG (arbitrarily set to 10, our CFG uses 5, being S, T, E, + and n).
- MAX_RULES: The maximal number of CFG production rules in a given CFG (arbitrarily set to 10, our CFG in Section 1 needs only 4).

Coding a CFG symbol object and its functions

Our first objective is to define a struct for the concept of a CFG symbol.

We would like the CFG symbol object/struct to consist of three attributes described below:

- Symbol: a char describing the symbolic representation of the CFG symbol, being set to either 'S', 'E', 'T', '+' or 'n'.
- Is_terminal: an int, with value 1 if the CFG symbol is terminal and 0 if it is a non-terminal.
- Is_start: an int, with value 1 if the CFG symbol is the starting symbol and 0 otherwise.

Question 2-A: Can you figure out the missing code in the struct CFGSymbol at the beginning of the code? Show your code in your report.

Question 2-B: For each possible CFG symbol in our CFG from Section 1 (i.e. 'S', 'E', 'T', '+' or 'n'), can you describe the values of the attributes set to each symbol?

Question 2-C: Let us now move on to the init_CFGSymbol() function. It will receive four parameters corresponding to a CFGSymbol object, a char text corresponding to the representation of this CFG symbol, an int is_terminal and an int is_start. This function should set the attributes of the CFGSymbol object using the other values provided, assigning them to their respective attributes in the CFGSymbol object. Show your code in your report.

Question 2-D: Let us now continue with the init_NonTerminal () function, the init_Terminal() function and the init_StartSymbol() function. Each of these functions will be used to initialize the 5 symbols used in our CFG in the main() function, as shown below. Figure out the missing code, reusing your function from Question 2-C accordingly. Show your code.

```
139  // Some test cases
140  int main(void) {
141      CFGSymbol S, E, T, plus, n;
142      CFGSymbol rhs0[1], rhs1[1], rhs2[3], rhs3[1], rhs4[1];
143      CFGProductionRule rule0, rule1, rule2, rule3, rule4;
144      CFG cfg;
145      CFGSymbol symbols[5];
146      CFGProductionRule rules[4];
147
148      // Initialize symbols and print them.
149      init_StartSymbol(&S, 'S');
150      init_NonTerminal(&E, 'E');
151      init_NonTerminal(&T, 'T');
152      init_Terminal(&plus, '+');
153      init_Terminal(&n, 'n');
```

If your functions in Questions 2-C and 2-D are correct, it should produce the following prints in the main() function.

```
--- Our Symbols ---
Symbol: S, Is Terminal: 0, Is Start: 1
Symbol: E, Is Terminal: 0, Is Start: 0
Symbol: T, Is Terminal: 0, Is Start: 0
Symbol: +, Is Terminal: 1, Is Start: 0
Symbol: n, Is Terminal: 1, Is Start: 0
```

Coding a CFG production rule object and its functions

Our next step is to define a struct for the concept of a CFG production rule.

We would like the CFGProductionRule object/struct to consist of the following three attributes described below:

- lhs: the left-hand side of the production rule, which consists of a single CFGSymbol.
- rhs: the right-hand side of the production rule, which consists of a sequence of CFGSymbols, defined as an array CFGSymbol elements, with size MAX_RHS.
- rhs_count: an int denoting the number of symbols in the right-hand side array.

Question 2-E: Can you figure out the missing code in the struct CFGProductionRule at the beginning of the code? Show your code in your report.

Question 2-F: We expect the CFGsymbol used in the lhs attribute to satisfy a certain property in terms of attribute values. Which property should all lhs symbols satisfy for a production rule to be valid?

Let us now move on to the function createProductionRule(). It will receive several parameters and should have the following behavior:

- It should check if lhs is a non-terminal symbol. It will display an error message that reads as “Invalid production rule, lhs is a terminal symbol.” and set the rhs_length attribute to -1 otherwise. If it is not a terminal symbol, it will assign it to the lhs attribute.

- It will then assign the sequence of symbols in CFGSymbol rhs[] to the production rule rhs attribute, and in the process, define the number of elements rhs_length.

Question 2-G: Show your code for your function createProductionRule() in your report.

Question 2-H: Consider the function printProductionRule(). This function should print a production rule. It can safely assumes that we will have a valid production rule already. It will then print the left-hand side symbol, then print " --> ", and finally iterate through the right-hand side symbols and print them.

Assuming you have correctly figured out both function, the code in the main, shown below will produce the expected print, as shown, if you have correctly figured out the function.

```
// Define the production rules and display them.
printf("\n--- Our Production rules ---\n");
rhs0[0] = n;
rule0 = createProductionRule(n, rhs0, 1);
rhs1[0] = E;
rule1 = createProductionRule(S, rhs1, 1);
rhs2[0] = E;
rhs2[1] = plus;
rhs2[2] = T;
rule2 = createProductionRule(E, rhs2, 3);
rhs3[0] = T;
rule3 = createProductionRule(E, rhs3, 1);
rhs4[0] = n;
rule4 = createProductionRule(T, rhs4, 1);
printProductionRule(rule1);
printProductionRule(rule2);
printProductionRule(rule3);
printProductionRule(rule4);
```

```
--- Our Production rules ---
Invalid production rule, lhs is a terminal symbol.
S --> E
E --> E+T
E --> T
T --> n
```

Coding a CFG object and its functions

Finally, let us move on to our CFG object, which will describe the CFG in its entirety. It should have the following attributes:

- symbols: An array of possible symbols, defined as an array of CFGSymbol, with size MAX_SYMBOLS.

- startSymbol: The start symbol of the CFG, as a single CFGSymbol.

- rules: An array of possible production rules, defined as an array of CFGProductionRule, with size MAX_RULES.

- symbol_count: An int, denoting the number of CFGSymbol elements in the symbols array.

- rule_count: An int, denoting the number of CFGProductionRule elements in the rules array.

Question 2-I: Show your code for your struct/object CFG in your report.

Question 2-J: Let us now focus on the init_CFG() function, which will be used to create a CFG. It receives a CFG struct, an array of CFGSymbols symbols, an array of CFGProductionRules rules, and counters for the lengths of these arrays. The function should simply assign each of these arrays and int values to the attributes of the CFG struct. Show your code in your report.

Question 2-K: Finally, focus on the function printCFG(). This function for printing the CFG as expected. It should display all the production rules in the format "(k) lhs --> rhs", with k in integer value starting from 1 and incrementing with each new production rule display.

If you have correctly figured out the functions in Question 2-I, 2J and 2-K, the code in the main() below will produce the expected prints.

Your init_CFG() function code may also, optionally, include additional checks:

- The CFGsymbol startSymbol should indeed be a start symbol. It should produce an error message is the CFGSymbol does not have a certain attribute value.
- The set of production rules should include at least one production rule the uses the start symbol as lhs.
- A more difficult challenge would consist of checking that the production rules given in the rules array can eventually transform the start symbol into to a sequence of terminal symbols only. This final check is difficult to resolve, but feel free to attempt it, for extra practice.

```
// Initialize the CFG and display it.
printf("\n--- Our CFG ---\n");
symbols[0] = S;
symbols[1] = E;
symbols[2] = T;
symbols[3] = plus;
symbols[4] = n;
rules[0] = rule1;
rules[1] = rule2;
rules[2] = rule3;
rules[3] = rule4;
int symbol_count = sizeof(symbols)/sizeof(symbols[0]);
int rule_count = sizeof(rules)/sizeof(rules[0]);
init_CFG(&cfg, symbols, symbol_count, S, rules, rule_count);
printCFG(cfg);
```

```
--- Our CFG ---
(1):   S --> E
(2):   E --> E+T
(3):   E --> T
(4):   T --> n
```

This concludes Task 2.

Task 3: Implementing a simple Tokenizer using RegEx

Description

In this section, we will discuss the implementation of a simple tokenizer, that we could use for the CFG framework we have built in Section 2.

You may open the code in the *Tokenizer.c* file. As you will see it starts with includes and defines, more specifically we have used two define statements to produce the following identifiers with the macro values:

- MAX_TOKENS: We will assume that the strings read by the Tokenizer can be processed as an array of maximum 20 Tokens.
- MAX_DIGITS: We will assume that the numbers in strings read by the Tokenizer consist of maximum 20 digits.

Start by copy pasting the code you have produced in the previous task for the CFGSymbol struct, init_CFGSymbol() function and init_Terminal().

Our objective in this task will be to write a Tokenizer that will read a sequence of digits and + operations, such as *"17+4+526"*. As a result, it will produce an array of CFGSymbols, whose symbols will later be printed on screen as:

Token(n) Token(+) Token(n) Token(+) Token(n)

Later on, we could try to match this array of CFGSymbols, by using our CFG to verify if the string *"17+4+526"* indeed consists of a valid arithmetic expression.

Question 3-A: Why are we importing the init_Terminal() function, but not the init_NonTerminal() function here?

Writing a simplified maximal munch algorithm for our Tokenizer

You may assume that the strings to tokenize will only consist of digits and '+' symbols, no white spaces or line break will appear in these strings either. We will also assume that we will not accept the notation "+7" for the integer number "7". Only positive numbers will be considered.

There is therefore no need to implement additional error features, in the event another character appears in the string. However, it would be a fantastic extra practice for you to amend your maximal munch algorithm to:

- Process whitespaces,
- And raise error messages in the case we have an unexpected character in the string.

Question 3-B: What would be the RegEx to use, to check whether a string "s" is a valid positive integer according to our description above?

Question 3-C: Given that the strings to tokenize will only consist of digits and '+' symbols, is it necessary in this case to use RegEx? What would be the pseudo-code for our maximal munch algorithm? Keep it as simple as possible.

In the *Tokenizer.c* file, there is a function tokenizeString(). This function receives:

- str: The string to tokenize, e.g. *"17+4+526"*.
- symbols: the array that will contain up to 20 CFGSymbols, after Tokenization.
- symbol_count: an int denoting the number of CFGSymbols in the symbols array.
- plus: the CFGSymbol for the terminal symbol "+".
- n: the CFGSymbol for the terminal symbol "n", corresponding to a number.

Your function should read the characters in the string str, one at a time, and tokenize it. It will then produce the correct sequence of CFGSymbols and store them in the array symbols. It will also update the value of symbol_count accordingly.

To help you, we have assembled all the test cases in the main() function, shown in the next page. It will produce the prints shown in the next page if your function is correct.

Question 3-D: Show your code for your tokenizeString() in your report.

```c
// Some test cases
int main(void) {
    CFGSymbol plus, n;
    CFGSymbol symbols1[MAX_TOKENS], symbols2[MAX_TOKENS];
    int symbol_count1 = 0;
    int symbol_count2 = 0;
    char str1[] = "17+4+526";
    char str2[] = "74++26+";

    // Initialize terminal symbols for Tokenizer.
    init_Terminal(&plus, '+');
    init_Terminal(&n, 'n');

    // Test case #1 for Tokenizer
    tokenizeString(str1, symbols1, &symbol_count1, &plus, &n);
    printf("--- Test case 1\nString: %s\nTokens: ", str1);
    for (int i = 0; i < symbol_count1; ++i) {
        printf("Token(%c) ", symbols1[i].symbol);
    }
    printf("\n");

    // Test case #2 for Tokenizer
    tokenizeString(str2, symbols2, &symbol_count2, &plus, &n);
    printf("--- Test case 2\nString: %s\nTokens: ", str2);
    for (int i = 0; i < symbol_count2; ++i) {
        printf("Token(%c) ", symbols2[i].symbol);
    }
    printf("\n");

    return 0;
}
```

```
--- Test case 1
String: 17+4+526
Tokens: Token(n) Token(+) Token(n) Token(+) Token(n)
--- Test case 2
String: 74++26+
Tokens: Token(n) Token(+) Token(+) Token(n) Token(+)
```

This concludes Task 3.

Task 4: Implementing a manual derivation engine for our CFG

Description

In this section, we will discuss the implementation of manual derivation engine for our CFG.

Our objective is to define several functions.

1. We will start by implementing a function, which will start the derivation. It then creates an array of CFGSymbols, which should contain only of a single symbol corresponding to the start symbol in our CFG.
2. We will then write a function which will use a production rule from the CFG and will replace a certain CFGSymbol in our array of CFGSymbols by another one, or by a sequence of other CFGSymbols.
3. We will finally write a function, whose objective is to check that the derivation is complete, and that the sequence of CFGSymbols in the array matches the one produced by the Tokenizer. If all CFGSymbols match, it will claim that the derivation was successful, and unsuccessful otherwise.

Manually figuring out the correct derivation.

Consider the CFG in Task 1 and the string “n+n+n”.

Question 4-A: What is a valid derivation for the string “n+n+n”? Mention the production rules you will be using, but also the position index corresponding to the symbol to be modified by each production rule.

Initializing the array of CFGSymbols

In this section and the next ones, we will be playing with the code in the *Derivation.c* file.

Let us first focus on the function startDerivation(). It receives three parameters:

- derivation: an array of CFGsymbols, initially empty.
- derivation_length: an int denoting the number of CFGSymbols in derivation, initially 0.
- cfg: our CFG object, which will be used for the derivation.

It should simply retrieve the start symbol used by the CFG and assign it as the first and only element in the derivation array. It will also update the derivation_length int accordingly.

Question 4-B: Show your code for the startDerivation() function in your report.

Applying a production rule on a given symbol in the derivation

Let us now focus on the second function applyProductionRule (). It receives several parameters:

- derivation: an array of CFGsymbols, hopefully, it is no longer empty.
- derivation_length: an int denoting the number of CFGSymbols in derivation.
- cfg: our CFG object, which will be used for the derivation.
- ruleIndex: the index of the production rule to be used to modify the sequence of symbols in derivation. Remember that the indexing for the production rules starts from 1 and that the rules are assembled in an array in the CFG object.
- position: a positional index denoting the position of a CFGSymbol in the array derivation. This symbol needs to be transformed following the production rule in the CFG.

This function will retrieve the production rule, located in the array of Production rules of our CFG at index ruleIndex. It will then apply in on the symbol currently located at position position in the array of CFGSymbols derivation.

It should perform all the needed checks (symbol in position corresponds to the lhs of production rule, will not break maximal length for sequence of tokens, etc.)

Eventually, will update the derivation and derivation_length variables, accordingly. Or do nothing, simply printing an error message if invalid.

You will have to shift symbols in the derivation array, in the case the lhs and rhs of the production rule do not consist of the same number of elements.

Question 4-C: Show your code for the applyProductionRule() function in your report.

Checking for a valid derivation

Finally, let us focus on the function checkDerivation(). It receives four parameters:

- derivation: an array of CFGsymbols, hopefully, it is no longer empty,
- derivation_length: an int denoting the number of CFGSymbols in derivation.
- tokens: an array of CFGsymbols corresponding to the one produced by the tokenizer.
- token_count: an int denoting the number of CFGSymbols in the tokens array.

This function will compare the elements in the derivation and tokens arrays. It should return 1, if all elements match, therefore indicating that the derivation was successful. Otherwise, it will return 0 (False). In the case of a mismatch, it should indicate the position of the first pair of symbols that are not matching. In positive outcomes, we will simply display “Derivation successful!”. Note that if derivation and tokens do not have the same number of symbols, no chance that there is a match. You might be able to skip some checks.

Question 4-D: Show your code for the checkDerivation () function in your report.

Final note before you start coding

In the *Derivation.c* file, we are providing the main() function, as well as a helper function called printArraySymbols(), which simply display the symbols in an array of CFGSymbols.

Your code should produce the following outcome, when executed. Take it slow, one step at a time, and do not hesitate to modify the main() if you want to check your functions are behaving correctly.

```
--- Test case 1
Token(S)
Token(E)
Token(E) Token(+) Token(T)
Token(E) Token(+) Token(T) Token(+) Token(T)
Token(T) Token(+) Token(T) Token(+) Token(T)
Token(n) Token(+) Token(T) Token(+) Token(T)
Token(n) Token(+) Token(n) Token(+) Token(T)
Token(n) Token(+) Token(n) Token(+) Token(n)
Derivation successful!

--- Test case 2
Token(S)
Rule cannot be applied at the given position.
Token(S)
Rule cannot be applied at the given position.
Token(S)
Token(E)
Token(E) Token(+) Token(T)
Token(E) Token(+) Token(T) Token(+) Token(T)
Token(T) Token(+) Token(T) Token(+) Token(T)
Token(n) Token(+) Token(T) Token(+) Token(T)
Token(n) Token(+) Token(n) Token(+) Token(T)
Token(n) Token(+) Token(n) Token(+) Token(n)
Token(+) Token(+) Token(n) Token(+) Token(n)
Derivation unsuccessful: Symbol mismatch at position 0.
```

This concludes Task 4 and Lab3.