50.051 Programming Language Concepts – Spring 2024

Atul Parida
1006184

Submitted 6th April 2024

Answers to tasks are given below. Each task is started on a new page (though for Task 1 that's a rather inefficient use of space, an antithesis to what C is all about):

Task 1:

A. The purpose of this CFG is to define the syntax of arithmetic expressions consisting of addition operations (+) between positive integers (represented by 'n' here). It lets us check whether a given string of symbols conforms to the CFG's grammar rules representing a valid arithmetic expression. This is used to validate and analyse the input code, convert it into tokens, and determine if it's a valid sequence or not within the compiler to make sure the input follows the compiler's syntax rules.

B. If a non-terminal symbol can be replaced by a sequence including that symbol itself, the production rule is recursive. Eg: in rule 2, the symbol E appears in both the LHS and RHS of the expression. It allows us to describe structures that can infinitely repeat or nest, such as expression like "n+n+n" or "n+n+n+n…" and so on, where a valid expression is repeated. Without it, parsing nested structures would be a lot more cumbersome.

Task 2:

A. The following code is present within “Task2.h”:

```c
// Struct for CFG symbols.
typedef struct {
    char symbol;
    int is_terminal;
    int is_start;
} CFGSymbol;
```

B. I’ve given the symbols and their characteristics in a table:

Symbol	Is it terminal?	Is it a Start Symbol?
S	Non-terminal (0)	Start symbol (1)
E	Non-terminal (0)	Not a start symbol (0)
T	Non-terminal (0)	Not a start symbol (0)
+	Terminal (1)	Not applicable (0)
n	Terminal (1)	Not applicable (0)

C. The following code is present within “Task2.c”:

```c
// Generic function to initialize a CFGSymbol.
void init_CFGSymbol(CFGSymbol* symbol, char text, int is_terminal, int is_start) {
    symbol->symbol = text;
    symbol->is_terminal = is_terminal;
    symbol->is_start = is_start;
}
```

D. The following code is present within "Task2.c”:

```c
// Specific initializers for different types of symbols (terminal symbol).
void init_Terminal(CFGSymbol* symbol, char text) {
    init_CFGSymbol(symbol, text, 1, 0);
}
```

E. The following code is present within “Task2.h”:

```c
// Struct for a production rule of our CFG.
typedef struct {
    CFGSymbol lhs;
    CFGSymbol rhs[MAX_RHS];
    int rhs_length;
} CFGProductionRule;
```

F. For a production rule to be valid, all symbols in the LHS should be non-terminal in nature.

G. The following code is present within “Task2.c”:

```c
CFGProductionRule createProductionRule(CFGSymbol lhs, CFGSymbol rhs[], int rhs_length) {
    CFGProductionRule rule;
    int i;

    // Check that lhs is not a terminal symbol (otherwise, problem)
    if (lhs.is_terminal) {
        printf("Invalid production rule, lhs is a terminal symbol.\n");
        rule.rhs_length = -1;
        return rule;
    }

    // Iterate through the right-hand side symbols and print them.
    // Initialize rhs_length to 0.
    // Later, count the actual length of rhs until the '\0' symbol is found.
    rule.lhs = lhs;
    for (i = 0; i < rhs_length; i++) {
        rule.rhs[i] = rhs[i];
    }
    rule.rhs_length = rhs_length;
    return rule;
}
```

H. The following code is present within “Task2.c”:

```c
void printProductionRule(CFGProductionRule rule) {
    int i;

    printf("%c", rule.lhs.symbol);
    printf(" --> ");
    for (i = 0; i < rule.rhs_length; i++) {
        printf("%c", rule.rhs[i].symbol);
    }
    printf("\n");
    return;
}
```

I. It outputs the correct output when run, as outlined in the Lab 3 instructions.

J. The following code is present within “Task2.h”:

```c
// Struct for the full CFG.
typedef struct {
    CFGSymbol symbols[MAX_SYMBOLS];
    CFGSymbol startSymbol;
    CFGProductionRule rules[MAX_RULES];
    int symbol_count;
    int rule_count;
} CFG;
```

K. The following code is present within "Task2.c":

```
void init_CFG(CFG* cfg, CFGSymbol symbols[], int symbol_count, CFGSymbol startSymbol,
CFGProductionRule rules[], int rule_count) {
    int i;

    for (i = 0; i < symbol_count; i++) {
        cfg->symbols[i] = symbols[i];
    }
    cfg->symbol_count = symbol_count;
    cfg->startSymbol = startSymbol;
    for (i = 0; i < rule_count; i++) {
        cfg->rules[i] = rules[i];
    }
    cfg->rule_count = rule_count;
    return;
}
```

L. The following code is present within "Task2.c":

```
void printCFG(const CFG cfg) {
    int i;

    for (i = 0; i < cfg.rule_count; i++) {
        printf("(%d):   ", i);
        printProductionRule(cfg.rules[i]);
    }
}
```

This outputs the correct print as required in the instructions to Lab 3.

Task 3:

A. We import the init_Terminal() function in the Tokenizer because it deals with terminal symbols, which are the focus of tokenization. The init_NonTerminal() function is not needed because the Tokenizer does not handle non-terminals or the CFG structure.

B. ^[1-9][0-9]*$ would suffice as a valid RegEx to our problem.

C. Since the character set is limited, we don't need a RegEx for tokenization.

Here's a simple pseudocode for the maximal munch algorithm specific to this problem:

```
tokens ← [ ]
string_buffer ← new StringBuffer(null)
for each character c in the input string:
  if c is a digit:
    string_buffer.append(c)
  else if string_buffer != null:
    converted_buffer ← tokenize(string_buffer)
    tokens.append(converted_buffer)
    string_buffer ← null
    string_buffer.append(c)
  else:
    raise an error ideally, or skip the character if we don't have error handling

if string_buffer != null:
  converted_buffer ← tokenize(string_buffer)
  tokens.append(converted_buffer)

return tokens
```

D. The following code is present within “Task3.c”:

```
void tokenizeString(char* str, CFGSymbol* symbols, int* symbol_count, CFGSymbol* plus,
CFGSymbol* n) {
    // Initialize indexing and symbol count.
    int i = 0;
    *symbol_count = 0;
    while (str[i] != '\0' && *symbol_count < MAX_TOKENS) {
        if (str[i] == plus->symbol) {
            symbols[*symbol_count] = *plus;
            *symbol_count += 1;
            i += 1;
        } else if (str[i] >= '0' && str[i] <= '9') {
            int start = i;
            while (str[i] >= '0' && str[i] <= '9' && i - start < MAX_DIGITS) {
                i += 1;
            }
            symbols[*symbol_count] = *n;
            *symbol_count += 1;
        } else {
            i += 1;
        }
    }
}
```

Task 4:

A. A valid derivation for “n+n+n” using our CFG from Task 1 is as follows:
   Rule 1, position 0: $S \rightarrow E$
   Rule 2, position 1: $E \rightarrow E + T$
   Rule 2, position 1: $E + T \rightarrow E + E + T$
   Rule 3, position 1: $E + E + T \rightarrow E + T + T$
   Rule 3, position 0: $E + E + T \rightarrow T + T + T$
   Rule 4, position 0: $T + T + T \rightarrow n + T + T$
   Rule 4, position 1: $n + T + T \rightarrow n + n + T$
   Rule 4, position 2: $n + n + T \rightarrow n + n + n$

B. The following code is present within “Task4.c”:

```c
void startDerivation(CFGSymbol* derivation, int* derivation_length,
CFG* cfg) {
    // Assign the start symbol of the CFG to the first position of
    the derivation array.
    derivation[0] = cfg->startSymbol;
    // Update the derivation length.
    *derivation_length = 1;
}
```

C. The following code is present within “Task4.c”:

```c
void applyProductionRule(CFGSymbol* derivation, int*
derivation_length, CFG* cfg, int ruleIndex, int position) {
    if (ruleIndex < 1 || ruleIndex > cfg->rule_count) {
        printf("Invalid rule index.\n");
        return;
    }
    CFGProductionRule rule = cfg->rules[ruleIndex - 1];
    if (position < 0 || position >= *derivation_length || derivation
    [position].symbol != rule.lhs.symbol) {
        printf("Rule cannot be applied at the given position.\n");
        return;
    }
    int new_length = *derivation_length + rule.rhs_length - 1;
    if (new_length > MAX_SYMBOLS) {
        printf("Applying the rule exceeds the maximum derivation
        length.\n");
        return;
    }
    for (int i = *derivation_length - 1; i > position; --i) {
        derivation[i + rule.rhs_length - 1] = derivation[i];
    }
    for (int i = 0; i < rule.rhs_length; ++i) {
        derivation[position + i] = rule.rhs[i];
    }
    *derivation_length = new_length;
}
```

D. The following code is present within “Task4.c”:

```c
// Function to check if we have a match in the derivation.
int checkDerivation(CFGSymbol* derivation, int derivation_length,
CFGSymbol* tokens, int token_count) {
    // If derivation and tokens do not have the same number of
    symbols,
    // no chance that there is a match.
    if (derivation_length != token_count) {
        printf("Derivation unsuccessful: Length mismatch, stopping
        early.\n");
        return 0;
    }

    // Otherwise, check all symbols one by one.
    // Stop early if two symbols do not match and return 0 (False).
    for (int i = 0; i < derivation_length; ++i) {
        if (derivation[i].symbol != tokens[i].symbol) {
            printf("Derivation unsuccessful: Symbol mismatch at
            position %d.\n", i);
            return 0;
        }
    }

    // Otherwise, return 1 (True).
    printf("Derivation successful!\n");
    return 1;
}
```