50.051 Programming Language Concepts

Simriti Bundhoo 1006281

Lab 3: Report

Task 1: Introduction to Context-Free Grammars

Question 1-A

It is checking if a positive number is being added to another positive number.

Question 1-B

When a production rule is recursive, it means that the rule can refer back to the non-terminal symbol it produces on its left-hand side within its right-hand side. This recursion allows the CFG to generate structured and potentially infinite sequences of symbols and to represent repetitive structures or patterns in the language being defined.

Task 2: Coding a simple CFG

Coding a CFG symbol object and its functions

Question 2-A

```
typedef struct {
    char symbol;
    int is_terminal;
    int is_start;
} CFGSymbol;
```

Question 2-B

S $\rightarrow$ symbol (is_start = 0)
E $\rightarrow$ symbol (is_terminal = 0)
T $\rightarrow$ symbol (is_terminal = 0)
+ $\rightarrow$ symbol (is_terminal = 0)
n $\rightarrow$ symbol (is_terminal = 1)

Question 2-C

```
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;
}
```

Question 2-D

```
// Specific initializers for different types of symbols (non-terminal symbol).
void init_NonTerminal(CFGSymbol* symbol, char text) {
    symbol->symbol = text;
    symbol->is_terminal = 0;
    symbol->is_start = 0;
}
```

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


// Specific initializers for different types of symbols (start symbol).
void init_StartSymbol(CFGSymbol* symbol, char text) {
    symbol->symbol = text;
    symbol->is_terminal = 0;
    symbol->is_start = 1;
}
```

Coding a CFG production rule object and its functions

Question 2-E

```c
typedef struct {
    CFGSymbol lhs;
    CFGSymbol rhs[MAX_RHS];
    int rhs_count;
} CFGProductionRule;
```

Question 2-F

They should be non-terminal.

Question 2-G

```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 == 1) {
        printf("Error: Left-hand side symbol must be a non-terminal symbol.\n");
        rule.rhs_count = -1;
        return rule;
    }
    else
    {
        rule.lhs = lhs;

        for (i = 0; i<rhs_length; i++) {
            rule.rhs[i] = rhs[i];
        }

        rule.rhs_count = rhs_length;
        return rule;
    }
}
```

Coding a CFG object and its functions

Question 2-I

```c
typedef struct {
    CFGSymbol symbols[MAX_SYMBOLS];
    CFGSymbol startSymbol;
    CFGProductionRule rules[MAX_RULES];
    int symbol_count;
    int rule_count;
} CFG;
```

Question 2-J

```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;
}
```

Task 3: Implementing a simple Tokenizer using RegEx

Question 3-A

The tokenizer deals only with terminal symbols (digits and the '+' symbol) directly from the input string. There are no non-terminal symbols or production rules involved in the tokenization process.

Writing a simplified maximal munch algorithm for our Tokenizer

Question 3-B

The RegEx expression is: $\hat{}[0-9]*\backslash+?[0-9]*\$$

Question 3-C

No, we can instead use a simple maximal munch algorithm without relying on RegEx.

Pseudocode

Initialize an empty array for tokens (tokens[n])
input_string ← " "

FOR char IN input_string:
    IF char == digit OR char == '+' THEN
        input_string.append(char)
    ELSE
        IF input_string is NOT empty THEN
            tokens.append(input_string)
            input_string ← " "

    IF input_string is NOT empty THEN
        tokens.append(input_string)
NEXT
RETURN tokens

Explanation:

The maximal munch algorithm initializes an empty 'tokens' array and 'input_string' string. It processes the input string character by character, appending digits or '+' symbols to 'input_string'. When a non-digit or non-'+' character is found, it adds the constructed token to 'tokens' if 'input_string' is not empty. After iterating through the string, any remaining 'input_string' is added to 'tokens'. Finally, the algorithm returns 'tokens', ensuring efficient tokenization by greedily forming tokens before encountering delimiters.

Question 3-D

```
void tokenizeString(char* str, CFGSymbol* symbols, int* symbol_count, CFGSymbol* plus, CFGSymbol* n) {
    // Initialize indexing and symbol count.
    int i = 0;
    *symbol_count = 0;

    // Browse through string on character at a time.
    // Stop if maximal number of tokens is reached.
    // (Will not happen for our test cases)
    while (*symbol_count < MAX_TOKENS && str[i] != '\0') {
        // If the current character is '+', add the plus CFGSymbol to the symbols array.
      if (str[i] == plus->symbol) {
            symbols[*symbol_count] = *plus;
            *symbol_count += 1;
            i += 1;
        }

      // If the current character is a digit, it might be the start of a number.
        // Mark the start of the number using the variable start.
      else if (str[i] >= '0' && str[i] <= '9') {
                // Keep incrementing i to find the end of the number.
         // Keep in mind the MAX_DIGITS constraint.
                int j = 0;
                while (str[i] >= '0' && str[i] <= '9' && j < MAX_DIGITS) {
                    i += 1;
                    j += 1;
                }
                // Add a single CFGsymbol n to represent the number.
                symbols[*symbol_count] = *n;
                *symbol_count += 1;

                // (Note: No need to decrement 'i' since we want to start reading the next character.)
        } else {
            // Handle non-digit and non-plus characters if necessary?
            i += 1;
        }
    }
}
```

Task 4: Implementing a manual derivation engine for our CFG

Manually figuring out the correct derivation

Question 4-A

- Start with the Start symbol S: S $\rightarrow$ E (Production Rule 1)
- Replace E with E + T: E $\rightarrow$ E + T (Production Rule 2)
- Repeat the previous step: E $\rightarrow$ E + T
- Replace the E with T: E $\rightarrow$ T (Production Rule 3)
- Replace each T with n: T $\rightarrow$ n (Production rule 4)

$\mathbf{S \rightarrow E \rightarrow E + T \rightarrow E + T + T \rightarrow T + T + T \rightarrow n + n + n}$

Initializing the array of CFGSymbols

Question 4-B

```
void startDerivation(CFGSymbol* derivation, int* derivation_length, CFG* cfg) {
    derivation[0] = cfg->startSymbol;
    *derivation_length = 1;
}
```

Applying a production rule on a given symbol in the derivation

Question 4-C

```
void applyProductionRule(CFGSymbol* derivation, int* derivation_length, CFG* cfg, int ruleIndex, int position) {
    // Validate the ruleIndex, checking that it corresponds to a production rule in the CFG.
   // (Remember that our indexing system for production rules starts at 1).
    int i;
    if (ruleIndex < 1 || ruleIndex > cfg->rule_count) {
        printf("Invalid rule index.\n");
        return;
    }

   // Retrieve production rule with index ruleIndex.
   // (Remember that our indexing system for production rules starts at 1).
    CFGProductionRule rule = cfg->rules[ruleIndex - 1];

    // Check if the position is valid for current derivation array.
   // Also check if the symbol at the position matches the lhs of the production rule.
    if (position < 0 || position >= *derivation_length || derivation[position].symbol != rule.lhs.symbol) {
        printf("Rule cannot be applied at the given position.\n");
        return;
    }

    // Calculate the new length of the derivation to check if it exceeds the limits.
    // COuld simply subtract 1 because we replace 1 symbol in lhs, always.
    int new_length = *derivation_length + rule.rhs_count - 1;
    if (new_length > MAX_SYMBOLS) {
        printf("Applying the rule exceeds the maximum derivation length.\n");
        return;
    }

    // Shift existing elements to make space for the new symbols from the rule's RHS.
    // (Probably a good idea to start from the end to avoid overwriting elements that need to be shifted)
    for (i=0; i < new_length; i++) {
        derivation[new_length - i] = derivation[*derivation_length - i];
    }

    // Insert the RHS symbols of the rule at the specified position.
    for (i = 0; i < rule.rhs_count; i++) {
        derivation[position + i] = rule.rhs[i];
    }

    // Update the derivation length.
    *derivation_length = new_length;
}
```

Checking for a valid derivation

Question 4-D

```
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.
```

```
    int i;

   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 (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;
}
```