Test 2

Programming Language Concepts

Due November ”16th”, 2020

Andrew Ballard

1. (20 points) Create code that allows you to create an ordered list of tokens. This code should take in a file as input and process that file for the following lexemes:
   * Perl style identifiers
   * Literals:
     + Java-Style string literals
     + C-Style integer literals
     + C-Style character literals
     + C-Style floating point literals
   * Non-Alphanumeric special symbols that denote ( at least two of which must be more than two characters ):
     + Addition
     + Assignment
     + Subtraction
     + Division
     + Multiplication
     + Increment
     + Decrement
     + Modulo Operator
     + Logical And
     + Logical Or
     + Logical Not
     + Open Code Block **–** Close Code Block
     + Open Function parameter
     + Close Function parameter

You may choose what ever symbol you represent for the special symbol but this must be explained in the comments with operation represents which symbol. Every type of lexeme defined must have a unique token equivalence. In this language every identifier must be followed by a non-alphanumeric character (excluding the character) too denote the end of the identifier. In this language every literal must be followed by white space or a special symbol to mark its end.

1

Programming Language Concepts Test 2 Due November 16ht, 2020

1. (9 points) Write three functions in C or C++: one that declares a large array statically, one that declares the same large array on the stack, and one that creates the same large array from the heap. Call each of the subprograms a large number of times (at least 100,000) and output the time required by each. Explain the results.

// Andrew Ballard

// Question 2

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// When called creates a large static array

void createArrayStatic(){

// Initializing large arrary statically

static int arrayStatic[12345678];

}

// When called creates a large array on the stack

void createArrayStack(){

// Initializing large array on the stack

int arrayStack[12345678];

}

// When called creates a large array from the heap

void createArrayHeap(){

// Initializing large array by using malloc to allocate memory for array from heap

int \*arrayHeap = (int \*) malloc(12345678 \* sizeof(int));

}

int main()

{

// Times to track start and end

struct timespec start, end;

// Timing static memory allocation 100,000 times

// Getting clock time at beginning of operation

clock\_gettime(CLOCK\_REALTIME, &start);

// Beginning the loop to call method 100,000 times

for (int i = 0; i < 100000; i++){

createArrayStatic();

}

// Getting clock time at the end of operation

clock\_gettime(CLOCK\_REALTIME, &end);

// Printing results for static array allocation

printf("Allocating memory for large static array 100,000 times complete.\n");

printf("Static Time: %d nanoseconds.\n", end.tv\_nsec - start.tv\_nsec);

// Timing memory allocation for large array on stack

// Getting clock time at beginning of operation

clock\_gettime(CLOCK\_REALTIME, &start);

// Beginning the loop to call method 100,000 times

for(int i = 0; i < 100000; i++){

createArrayStack();

}

// Getting clock time at the end of operation

clock\_gettime(CLOCK\_REALTIME, &end);

// Printing results for stack array allocation

printf("\nAllocating memory for large stack array on stack 100,000 times complete.\n");

printf("Stack Time: %d nanoseconds.\n", end.tv\_nsec - start.tv\_nsec);

// Timing memory allocation for large array on heap

// Getting clock time at beginning of operation

clock\_gettime(CLOCK\_REALTIME, &start);

// Beginning the loop to call method 100,000 times

for(int i = 0; i < 100000; i++){

createArrayHeap();

}

// Getting clock time at the end of operation

clock\_gettime(CLOCK\_REALTIME, &end);

// Printing results for stack array allocation

printf("\nAllocating memory for large stack array on heap 100,000 times complete.\n");

printf("Heap Time: %d nanoseconds.\n", end.tv\_nsec - start.tv\_nsec);

}

1. (11 points) Write an EBNF or CFG that while handle prefix/preorder Arithmetic Operations (addition, subtraction, multiplication, division, modulo) with the proper order of operations? What all types of parsers can be used to show the syntax for this? Justify your answer.

Arithmetic -> Arithmetic Operation | integer | (Arithmetic Operation)

Operation -> \* | / | + | - | %

The types of parsers that can be used to show the syntax for this are LL parsers and recursive decent parsers. An LL parser means that the parser is a left-to-right scan of the input and that a leftmost derivation is generated. A recursive descent parser is a type of LL parser that builds the parse tree in preorder/top down. Each node is visited before its branches are followed.

1. (10 points) What features of the compilation process allow us to determine the reference environment for any given line of code in the program. Answer this question for both dynamic and static scoping? Does the type of scoping change this answer? Explain why?

The part of the compilation process that allows us to determine the reference environment for any given line of code in the program is namely the scoping. Dynamic scoping is less widely used than static scoping because static scoping is easier to understand the context just by looking at a block of code. However, dynamic scoping is concerned more with how the program executes than how it is written. Each identifier has a global stack of bindings and the occurrence of an identifier is searched in the most recent binding. On the other hand, static scoping (also called lexical scoping) always refers to it top level environment. Static scoping is binding names to nonlocal variables known as static. The scope of a variable here is determined prior to execution and makes for easier code readability. The search begins in the subprogram containing the reference to the variable, then continues to the enclosing program, then to al the static ancestors of the subprogram.

1. (10 points) Detail how you would go about adding reserved words into the problem where you are designing your own lexical analyzer? How would you have to change your code? What would you have to add to let users choose a reserve word as an identifier?

When designing a lexical analyzer, reserved words should be given a higher rule priority over other tokens. You would have to change your code to implement this by throwing an error if the user tries to use a reserved work as an identifier, for example. In order to allow a user to choose a reserved word as an identifier, you could change the priority of keywords to be lower than identifiers so a user’s identifier choice is “more important” than a reserved word.

1. (20 points) Write a recursive decent algorithm for a java while statement, a Javas if statement , an logical/mathematical expression based on the rules you created in your lexical analyzer, and an mathematical assignment statement , where statement may be an empty function. Supply the EBNF rule for each.

EBNF Rules:

WHILE:

<while\_stmt> -> WHILE ‘(‘(<arithm\_expr> | <logical\_expr)’)’

<block><block> -> <stmt> | ‘{‘<stmt> {<stmt>} ‘}’

IF:

<if\_stmt> -> if (<boolean\_expr>) < stmt> {else <stmt}

LOGICAL/MATHEMATIC EXPRESSION:

<expression> -> <term> {(+|-) <term>}

<term> -> <factor> {(\*|/) <factor>}

<factor> -> ‘(‘ <expression> ‘)’ | name | number

ASSIGNMENT STATEMENT:

<assign> -> <id> = <expr>

<id> -> A|B|C

<expr> -> <expr> (\*|+|-|/) <expr> | <id> | (<expr>)

// JAVA WHILE STATEMENT ALGORITHM:

void whilestmt() {

/\* Be sure the first token is 'when' \*/

if (nextToken != WHILE\_CODE)

error();

else {

/\* Call lex to get to the next token \*/

lex();

/\* Check for the left parenthesis \*/

if (nextToken != LEFT\_PAREN)

error();

else {

/\* Parse the Boolean expression \*/

boolexpr();

/\* Check for the right parenthesis \*/

if (nextToken != RIGHT\_PAREN)

error();

else {

statement();

} /\* end of if (nextToken == ELSE\_CODE ... \*/ } /\* end of else of if (nextToken != RIGHT ... \*/ } /\* end of else of if (nextToken != LEFT ... \*/ } /\* end of else of if (nextToken != WHILE\_CODE ... \*/ } /\* end of ifstmt \*/

// JAVA IF STATEMENT:

/\* Function ifstmt Parses strings in the language generated by the rule: <ifstmt> -> if (<boolexpr>) <statement> [else <statement>] \*/

void ifstmt() {

/\* Be sure the first token is 'if' \*/

if (nextToken != IF\_CODE)

error();

else {

/\* Call lex to get to the next token \*/

lex();

/\* Check for the left parenthesis \*/

if (nextToken != LEFT\_PAREN)

error();

else {

/\* Parse the Boolean expression \*/

boolexpr();

/\* Check for the right parenthesis \*/

if (nextToken != RIGHT\_PAREN)

error();

else {

/\* Parse the then clause \*/

statement();

/\* If an else is next, parse the else clause \*/

if (nextToken == ELSE\_CODE) {

/\* Call lex to get over the else \*/

lex();

statement();

} /\* end of if (nextToken == ELSE\_CODE ... \*/ } /\* end of else of if (nextToken != RIGHT ... \*/ } /\* end of else of if (nextToken != LEFT ... \*/ } /\* end of else of if (nextToken != IF\_CODE ... \*/ } /\* end of ifstmt \*/

// HELPERS

/\* expr Parses strings in the language generated by the rule: -> {(+ | -) } \*/

void expr() {

printf("Enter \n");

/\* Parse the first term \*/

term();

/\* As long as the next token is + or -, get the next token and parse the next term \*/

while (nextToken == ADD\_OP || nextToken == SUB\_OP) {

lex();

term();

}

printf("Exit \n");

} /\* End of function expr \*/

/\* term Parses strings in the language generated by the rule: -> {(\* | /) ) \*/

void term() {

printf("Enter \n");

/\* Parse the first factor \*/

factor();

/\* As long as the next token is \* or /, get the next token and parse the next factor \*/

while (nextToken == MULT\_OP || nextToken == DIV\_OP) {

lex();

factor();

}

printf("Exit \n");

} /\* End of function term \*/

/\* factor Parses strings in the language generated by the rule: -> id | int\_constant | ( <expr) \*/

void factor() {

printf("Enter \n");

/\* Determine which RHS \*/

if (nextToken == IDENT || nextToken == INT\_LIT)

/\* Get the next token \*/

lex();

/\* If the RHS is ( <expr>), call lex to pass over the left parenthesis, call expr, and check for the right parenthesis \*/

else {

if (nextToken == LEFT\_PAREN) {

lex();

expr();

if (nextToken == RIGHT\_PAREN)

lex();

else

error();

} /\* End of if (nextToken == ... \*/

/\* It was not an id, an integer literal, or a left parenthesis \*/

else

error();

} /\* End of else \*/

printf("Exit <factor>\n");;

} /\* End of function factor \*/

int lex() {

lexLen = 0;

getNonBlank();

switch (charClass) {

/\* Parse identifiers \*/

case LETTER:

addChar();

getChar();

while (charClass == LETTER || charClass == DIGIT) {

addChar();

getChar();

}

nextToken = IDENT;

break;

/\* Parse integer literals \*/

case DIGIT:

addChar();

getChar();

while (charClass == DIGIT) {

addChar();

getChar();

}

nextToken = INT\_LIT;

break;

/\* Parentheses and operators \*/

case UNKNOWN:

lookup(nextChar);

getChar();

break;

/\* EOF \*/

case EOF:

nextToken = EOF;

lexeme[0] = 'E';

lexeme[1] = 'O';

lexeme[2] = 'F';

lexeme[3] = 0;

break;

} /\* End of switch \*/

printf("Next token is: %d, Next lexeme is %s\n", nextToken, lexeme);

return nextToken;

} /\* End of function lex \*/

These functions can be found inside our class textbook :)

1. (10 points) Given the natural constraints of an RDA explain how you would go about the creation of a Statement function in your RDA that would allow statement to either be a while statement, an if statement or an assignment statement.

A recursive descent function’s biggest problem as an LL parser is left recursion. As long as the statement function has all of the information needed, correct parenthesis, and correct scoping using brackets, the statement will be able to be either a while, if, or an assignment statement. The left recursion problem that exists within top-down parsers is not present with bottom-up parsers. Removing direct and indirect left recursion could solve this problem in this case.

1. (10 points) Perl allows both static and a kind of dynamic scoping. Write a Perl program that uses both and clearly shows the difference in effect of the two. Explain clearly the difference between the dynamic scoping described in this chapter and that implemented in Perl.

ALSO ATTACHED AS Q8.txt

# Assigning initial value to int variable

$int = 0;

# Outputing static int results to user

print "static int: " , staticScoping(), "\n";

# Outputting dynamic int results to user

print "dynamic int: " , dynamicScoping();

# function to return int variable

sub function{

return $int;

}

# Assigning value to int using dynamic scoping

sub dynamicScoping {

local $int = 1;

return function();

}

# Assigning value to int using static scoping

sub staticScoping {

my $int = 1;

return function();

}