Compiler Design Laboratory 8

Name: Sayan Paul
Roll: 21CS8056

Question:

Provide regular expressions to characterize the following lexical items:

Identifiers: alphanumeric strings starting with an alphabet, and can also contain special character '_' where you allow it is up to you, but state your decisions in your documentation. Examples x, ab_2y, c2 etc.
Numbers, which include integers, fixed point and floating point numbers. Leading zeros and redundant trailing zeros are disallowed; e.g., 002 is invalid, 2.00 is invalid but 2.0 is valid.
Real numbers can be represented in the E (exponential) format as well (e.g. 2.0E-2 is valid and represents 0.002). Decisions whether 2.0E+2 is valid are up to you.
white space (sequences of blank, tab, newline)
Arithmetic operators: +, -, /, *, div, mod
Logical constants: true, false
Logical operators: not, and, or
Comparison operators: <=, >=, <, > , =
Parentheses: (,)

Generate tokens from the regular expression implementing the Maximal Munch Tokenizer Algorithm.

Devise a table driven parser, based on the LL(1) parsing table developed in class, for the following grammar.
E -> E+E | E*E | E-E | E div E | E / E | E mod E | (E)
B -> E <= E | E >= E | E < E | E > E | E = E
B -> B and B | B or B | not B

Your recursive descent parser should ***interface*** with the lexical tokenizer developed as above to find out whether certain identifiers are reserved words of a certain programming Language (Like int in 'C'), when it reads an identifier.
One Idea may be to maintain a separate table (with a suitable data structure that facilitates search like a binary search tree of global reserved words) for reserved words.
The tokenizer can handle all identifiers simply as identifiers, but provide additional procedures to determine if an identifier is a globally reserved word or not.
Then when the parser reads an identifier it queries the tokenizer as to whether the identifier is one or the other.

To implement this, your tokenizer program would should additional routines to manipulate tokens and return most relevant information to the parser upon query from the parser.

Note: You may have to hack this grammar into a suitable form before it is LL(1).

Note: The grammar is deliberately ambiguous. You need to disambiguate it in the standard way, and give precedence according to the following conventions:

Parentheses bind tightest.
In expressions all binary operations are left associative.
*, /, div, mod bind tighter than +, -.
Arithmetic operators bind tighter than comparison operators <=, >=, <, > , =.
The comparison operators are non-associative and of equal precedence.
The comparison operators bind tighter than logical operators not, which bind tighter than and, which binds tighter than or.

You are only expected to provide skeleton code in C that would express your setting up of relevant data structures and routines that facilitates this possible interaction between your lexer and parser. Complete working programs are not required as no sample input or a specific programming language syntax is provided. Pls submit a write up explaining how you expect your code to work.

Code Skeleton:

```
/*
    Author: Dhruba Sinha
    Roll no: 21CS8051
*/


#include <stdio.h>
#include "defs.h"       //header for structure definitions required by
lexical-analyzer and LL(1)-parser


/*/////////////////////////////////////////
    global data-structures required by lexical-analyzer and LL(1)-parser
/////////////////////////////////////////*/

SymbolTable symtable;   //global symboltable for storing info about
identifiers like variables, functions
                        //This is an array of structures holding special
infos for identifiers like virtual-memory address , value etc.
                        //structure should be properly defined in defs.h


Regex regexlist;        //global declaration for list of structures
containing regex and corresponding token-class
                        // for example [0-9]+ for token-class "int"
                        //its structure should be defined in defs.h

NFA  ThompsonNFA;       //declaration for thompson-nfa , should be
initialized by init_lexical-analyzer(). structure should be defined in
defs.h
DFA  lexer-dfa;         //declaration for dfa , should be initialized by
init_lexical-analyzer(). structure should be defined in defs.h
                        //this DFA enables us to apply maximal-munch
tokenizing algorithm in one pass without backtracking, ensuring O(n) time

string program;         //program string to be parsed
```

```
int pos, len;            //pos = current position in the program-string,
where program[pos .. len - 1] is still to be tokenized
                         //len = length of the program string

BST reserverd_keywords; //global declaration for a balanced-BST (ex-
red-black-tree) of globally reserved keywords like 'while', 'for' etc.
                         //the balanced-BST ensures O(log n) search time
                         //Node structure should be defined in defs.h

Grammar g;               //global declaration for CFG which should be
imposed on the given code segment
                         //its structure should be defined in defs.h, with
proper feilds like V, T, P, S

int nullable[];          //declaration of nullable array,
                         //a non-terminal N is nullable iff nullable[N] =
1;

List FIRST[], FOLLOW[]; //declarations for FIRST and FOLLOW sets for
non-terminals in g.V

Table predict;           //predictor table for LL(1)-predictive parser
                         //predict[N][t] gives the production NO when
top-of-the symbol-stack is 'N' and current-read pointer is on 't'




/*////////////////////////////////////////////////
    routines for lexical-analyzer
/////////////////////////////////////////////////*/

void init_lexical-analyzer() {                          //initializer routine
for lexical-analyzer, called once from main()
    input_regexlist();                            //reads regex from user
    build_reserverd_keywords();                   //builds BST for
reserved_keywords
    buildNFA();                                   //builds ThompsonNFA from
regexlist
```

```
    convert_nfa_to_dfa();                     //builds lexer-dfa from
ThompsonNFA by subset-construction and minimization
    pos = 0;
    len = progrm.length();
}


Token* next_token() {                                  //returns the next
token using the maximal-munch algorithm
    Token t = new Token;
    if (pos == len) return NULL;                             //already at
the end

    /*
        maximal-munch algorithm here gives the token-class value
        this algo uses the lexer-dfa to efficiently find a match
    */


    if (token-class value is (-1)) {                            //didn't
find any match
        t->val = -1;                                            //-1 for
error
    }
    else if (matched_substring is in reserverd_keywords) {       //found
match is a keyword
        t->val = reserverd_keywords[matched_substring];
    }
    else {
        t->val = token-class value;
        t->attribute = token-class attribute;                    //
required for identifiers to locate symbol-table entry
    }

    pos += matched_substring.length();                          //update
read-pointer pos
    return t;
}
```

```
/*//////////////////////////////////////////////////
    routines for LL(1)-parser
//////////////////////////////////////////////////*/

void init_parser() {                                 //initializer routine
for parser, called once from main()
    input_grammar();                           //take grammar input and form the
grammar struct g accordingly;
    remove_ambiguity();                       //remove ambiguity from the gr by
properly defining precedence
    remove_left_recursion();                  //removes left recursion from g as
we r going to work with top-down parser
    calculate_nullable();                     //fills up the nullable array
    calculate_FIRST();                        //calculates FIRST set for all
non-terminals
    calculate_FOLLOW();                       //calculates FOLLOW set for all
non-terminals
    build_predictor_table();                  //builds predictor table predict
based on FIRST and FOLLOW
}


int parse() {                                        //main parsing
routine, returns 0 if no parsing error else returns -1
    Stack sym_stack;                      //symbol stack
    sym_stack.push(g.S);                  //push the start symbol
    Token* t = next_token();
    while (t != NULL) {                           //loop till end
        if (t->val == (-1)) return -1;        //error while tokenizing.. :(

        else if (sym_stack.top() is terminal) {       //top is a terminal.
remember that terminasl(g.T) should always be one of token classes.
            if (sym_stack.top() != t->val) return -1      //syntax-error
            else {                            //match!!!!! :) consume terminal
                sym_stack.pop();
                t = next_token();
            }
        }
        else {                                       //top is a
non-terminal
            int production_no = predict[sym_stack.top()][t->val];
            if (production_no == (-1)) return -1;  //no-valid production
..... parsing error :(
            else {
                sys_stack.pop();                     //pop the non-terminal
                for (int i = g.P[production_no].body.length() - 1; i >= 0;
i--) sys_stack.push(g.P[production_no].body[i]);   //push the body of the
production in reverse order
            }
        }
    }

    if (sym_stack.empty()) return 0;                    //parsing
successfull :)
    else return -1;                                     //error .. :(
}


/*/////////////////////////////////////
    main routine
//////////////////////////////////////*/

int main()
{
    input_program();                             //read the program string
into program
    init_lexical-analyzer();                     //initialize lexical-analyzer
and its related data-structures
    init_parser();                               //initialize parser and its
related data-structures

    if (parse() == 0 /* try to parse */) fprintf(stdout, "successfully
parsed\n");
    else fprintf(stderr, "error while parsing\n");

    return 0;
}
```

Explanation:

Header Inclusions:

The inclusion of <stdio.h> suggests that the program employs standard input-output operations, while "defs.h" implies a modular design where essential structures and definitions are separated from the main program logic. This modular approach can enhance readability, maintainability, and scalability of the codebase.

Global Data Structures:

The declaration of global data structures signifies the backbone of the compiler's functionality.

- SymbolTable serves as a repository for identifier-related information such as variable names, functions, and their attributes like virtual memory addresses and values.
- Regex holds a collection of regular expressions and their corresponding token classes, crucial for lexical analysis.
- NFA represents the Non-deterministic Finite Automaton, a foundational concept in lexical analysis.
- DFA is the Deterministic Finite Automaton, derived from NFA, enabling efficient lexical analysis through maximal-munch tokenization.
- string stores the program to be parsed, facilitating subsequent parsing stages.
- pos and len are vital for tracking the current position and length of the program string during lexical analysis.
- BST implements a balanced binary search tree, acting as a fast lookup structure for reserved keywords in the language.
- Grammar encapsulates the language's context-free grammar, pivotal for LL(1) parsing.
- nullable, FIRST, FOLLOW, and Table collectively represent essential data structures and sets utilized in LL(1) parsing algorithms.

Routines for Lexical Analyzer:

- init_lexical_analyzer() initiates the lexical analysis phase. It involves several crucial steps:
    - Reading regular expressions from the user, defining token classes.
    - Building a binary search tree to efficiently store and retrieve reserved keywords.
    - Constructing an NFA from the provided regular expressions, laying the foundation for pattern matching.
    - Transforming the NFA into a DFA via subset construction and minimization, optimizing tokenization efficiency.
    - Initializing the position and length variables for subsequent tokenization of the program string.
- next_token() implements the maximal-munch algorithm, a fundamental strategy in lexical analysis. It scans the program string, matching the longest possible token using the lexer DFA, and returns the corresponding token.

Routines for LL(1)-Parser:

- init_parser() sets up the LL(1) parsing environment. This involves a series of critical tasks:
    - Taking input of the grammar for the target language.
    - Removing ambiguity and left recursion from the grammar, ensuring deterministic parsing.
    - Calculating nullable, FIRST, and FOLLOW sets for non-terminals, foundational for LL(1) parsing algorithms.
    - Building the LL(1) parsing table, a crucial data structure that guides parsing decisions.
- parse() serves as the core parsing routine. It employs a symbol stack to perform LL(1) parsing, traversing the program string based on the parsing table. The function returns 0 upon successful parsing and -1 if parsing errors occur.

Main Routine:

The main() function orchestrates the execution flow of the compiler frontend. It follows a systematic approach:

- Reads the program string into memory, preparing it for lexical analysis and parsing.
- Initializes the lexical analyzer and parser, setting up their respective data structures and environments.
- Attempts to parse the program using the LL(1) parsing algorithm.
- Outputs ``successfully parsed" to the standard output if parsing completed without errors, or "error while parsing" to the standard error if issues arise during parsing.

Flow of the Code:

Reading the Program String:

The main() function starts by invoking input_program(), which reads the above program as a string. This string is stored in the global variable program, with len capturing its length.

Initializing the Lexical Analyzer:

init_lexical_analyzer() is called next. This function performs several key steps:

- input_regexlist() prompts the user (or reads from a predefined source) to input the regular expressions that define the syntax of numbers, identifiers, keywords, and symbols in the language. For our program, this might include regex patterns for begin, end, identifiers like x and y, assignment =, numeric literals, and arithmetic operators like +.
- build_reserverd_keywords() constructs a BST with reserved keywords (begin, end in our case), ensuring efficient lookups.
- buildNFA() and convert_nfa_to_dfa() transform the input regex patterns into a DFA for tokenizing the input program.

Lexical Analysis (Tokenization):

As the parsing process begins, parse() calls next_token() repeatedly to tokenize the input program. Here's how it might process the first few tokens:

- Identifies begin as a keyword token.
- Skips whitespace.
- Identifies x as an identifier token and assigns it a reference in the SymbolTable.
- Recognizes = as an assignment operator token.
- Identifies 3 as a numeric literal token.
- ...and so on, until the entire program is tokenized.

Initializing the LL(1)-Parser:
Before the actual parsing starts, init_parser() sets up necessary structures for **LL(1) parsing:**

- input_grammar() defines the grammar rules for the language. For example, it might specify that a statement can be an assignment, and an assignment takes the form of identifier = expression;.
- It processes the grammar to remove ambiguities, left recursion, calculates nullable, FIRST, and FOLLOW sets for non-terminals, and finally constructs the predictive parsing table.

LL(1) Parsing:
With the program tokenized and the parser initialized, the parse() function iteratively consumes tokens:

- It starts with the begin token, matching it against the start symbol's expected token, progressing through the grammar rules.
- As it encounters each token (x, =, 3, ;, y, =, x, +, 2, ;, end), it checks the parsing table to determine which grammar rule to apply next.
- If at any point, a token does not match the expected token (based on the parsing table and current stack top), it signifies a syntax error, and parsing fails (return -1).
- If all tokens match expected patterns and the symbol stack is emptied correctly, the program is considered successfully parsed (return 0).

Output Result:
Depending on whether the parsing was successful or not, main() outputs either "successfully parsed" or "error while parsing" to the user.