CS323 Documentation

About 2 pages

1. **Problem Statement**

   With the lexical analyzer constructed, the next step is to construct a syntax analyzer. This part of the compiler recognizes the structure of the source code using a finite set of rules called productions. The SA will utilize a Top-Down Parser, like the Recursive Descent Parser (RDP), to parse the tokens and fit them to the productions to determine whether the code properly follows the grammatical rules of RAT24S. For RDP to function optimally, left recursion and backtracking must be removed, so the 29 rules of RAT24S must be checked and rewritten if necessary. The parser will use the lexer() function from assignment 1 to acquire the tokens and lexemes from the source code file. Then, as the parser calls the lexer for a token and the production rules to analyze the token, it should print the tokens, lexemes, and production rules used for the token. If there is a syntax error in the source code, the parser should print an error message, which says the token, the lexeme, the line number, and the error type, and then it may exit the program. A fully functional syntax analyzer should be able to parse the entire program if it is syntactically correct.

2. **How to use your program**
   1. Navigate to the project directory.
   2. Open command prompt from the directory.
   3. Type: main2.exe
   4. You will be prompted to enter the name of a source code text file.
      a. Please type the name of the text file in the project directory that holds the RAT24S code (or junk code) that you wish to use.
      b. Example: Type: syntax_test_1.txt
   5. You will be prompted to enter another text file name to write the output.
      a. Please either *add a text file yourself* to the project directory for your own test files, *or use the file (jout.txt) provided for you*.
      b. Example: Type: synoutput1.txt

3. **Design of your program**

   The main file takes user input for the source and output files, and opens those files with a file instream and a write file pointer respectively. Then, it calls the first rule function RAT24S() in the SA.cpp file and passes a reference to the file instream and the write file pointer to the output file. Each function in SA.cpp is written according to the modified production rules for RAT24S, and they call each other, passing the file instream and the write file pointer. When a token has been matched to a token in a production rule, the rule calls the lexer() function in LA.cpp, passing the next char, the instream, and a reference to a global integer in SA.cpp which tracks the line number, so the lexer can increment it if it sees a newline character. The return value of the lexer is stored in a global std::pair<string, string> variable which stores the lexeme and token, so any function can access them to continue the top-down parsing of the tokens. A global bool constant is used to specify whether the tokens, rules, and errors should be written to the output file.

MODIFIED RULES:

R1. <Rat24S> ::= $ <Opt Function Definitions> $ <Opt Declaration List> $ <Statement List> $
R2. <Opt Function Definitions> ::= <Function Definitions> | <Empty>
R3 <Function Definitions> ::= <Function> <Function Definitions'>
R3.5 <Function Definitions'> ::= <Function Definitions> | ε
R4. <Function> ::= function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>
R5. <Opt Parameter List> ::= <Parameter List> | <Empty>
R6 <Parameter List> ::= <Parameter> <Parameter List'>
R6.5 <Parameter List'> ::= , <Parameter List> | ε
R7. <Parameter> ::= <IDs > <Qualifier>
R8. <Qualifier> ::= integer | boolean | real
R9. <Body> ::= { < Statement List> }
R10. <Opt Declaration List> ::= <Declaration List> | <Empty>
R11 <Declaration List> ::= <Declaration> ; <Declaration List'>
R11.5 <Declaration List'> ::= <Declaration List> | ε
R12. <Declaration> ::= <Qualifier > <IDs>
R13 <IDs> ::= <Identifier> <IDs'>
R13.5 <IDs'> ::= , <IDs> | ε
R14 <Statement List> ::= <Statement> <Statement List'>
R14.5 <Statement List'> ::= <Statement List> | ε
R15. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While>
R16. <Compound> ::= { <Statement List> }
R17. <Assign> ::= <Identifier> = <Expression> ;
R18 <If> ::= if (<Condition>) <Statement> <If'>
R18.5 <If'> ::= endif | else <Statement> endif
R19 <Return> ::= return <Return'>
R19.5 <Return'> ::= ; | <Expression>;
R20. <Print> ::= print (<Expression>);
R21. <Scan> ::= scan (<IDs>);
R22. <While> ::= while (<Condition>) <Statement> endwhile
R23. <Condition> ::= <Expression> <Relop> <Expression>
R24. <Relop> ::= == | != | > | < | <= | =>
R25 <Expression> ::= <Term> <Expression'>
R25.5 <Expression'> ::= + <Term> <Expression'> | - <Term> <Expression'> | ε
R26 <Term> ::= <Factor> <Term'>
R26.5 <Term'> ::= * <Factor> <Term'> | / <Factor> <Term'> | ε
R27. <Factor> ::=- <Primary> | <Primary>
R28 <Primary> ::= <Identifier><Primary'> | <Integer> | (<Expression>) | <Real> | true | false
R28.5 <Primary'> ::= (<IDs>) | ε
R29. <Empty> ::= ε

4. Any Limitation

None

5. Any shortcomings

None