CS323 Documentation

Part I: Problem Statement

The assignment is to write:

1. Symbol table handling
2. Generating an assembly code for the simplified version of Rat18F.

Part II: How to Use Our program

- Please rename your test file as "TestFile.txt"
- Please type the following command to run our executable file:

```
$./MacExec TestFile.txt
```

- To make things clarity, our program will print out three output text files as the results of running our program: *Lexical_Analysis_Result.txt*, *Syntactic_Analysis_Result.txt* and *Symbol_And_Instruction_Tables.txt*

  Note: If you run our program with two continuous tries, i.e. you run the program once and run the second time right after that; The result from the second test file might be appended into the first test file's.

  Advise: please delete the text file results after examining each run to receive an appropriate behavior of our program.

Part III: The Design of Our Program

1. We will rewrite the following original grammars of Rat18F to remove any left recursion (use left factorization if necessary).

A) Original Syntax rules : The following BNF describes the Rat18F.

R1. <Rat18F> ::= <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$
R2. <Opt Function Definitions> ::= <Function Definitions> | <Empty>
R3. <Function Definitions> ::= <Function> | <Function> <Function Definitions>
R4. <Function> ::= function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>
R5. <Opt Parameter List> ::= <Parameter List> | <Empty>
R6. <Parameter List> ::= <Parameter> | <Parameter> , <Parameter List>
R7. <Parameter> ::= <IDs > : <Qualifier>
R8. <Qualifier> ::= int | boolean | real
R9. <Body> ::= { < Statement List> }
R10. <Opt Declaration List> ::= <Declaration List> | <Empty>
R11. <Declaration List> := <Declaration> ; | <Declaration> ; <Declaration List>
R12. <Declaration> ::= <Qualifier > <IDs>
R13. <IDs> ::= <Identifier> | <Identifier>, <IDs>
R14. <Statement List> ::= <Statement> | <Statement> <Statement List>
R15. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While>
R16. <Compound> ::= { <Statement List> }
R17. <Assign> ::= <Identifier> = <Expression> ;
R18. <If> ::= if (<Condition>) <Statement> ifend |
if (<Condition>) <Statement> else <Statement> ifend
R19. <Return> ::= return ; | return <Expression> ;
R20. <Print> ::= put (<Expression>);
R21. <Scan> ::= get (<IDs>);
R22. <While> ::= while (<Condition>) <Statement> whileend
R23. <Condition> ::= <Expression> <Relop> <Expression>
R24. <Relop> ::= == | ^= | > | < | => | =<
R25. <Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term>
R26. <Term> ::= <Term> * <Factor> | <Term> / <Factor> | <Factor>
R27. <Factor> ::= - <Primary> | <Primary>
R28. <Primary> ::= <Identifier> | <Integer> | <Identifier> (<IDs>) | (<Expression>) | <Real> | true | false
R29. <Empty> ::= ε

B) **Production rules after removing left recursion and backtracking, and adding semantic meaning in some selected functions:**

BOLD words are nonterminating characters, underline statements are injected functions

R1. **RAT** => **OFD $$ ODL SL $$**
R2. **OFD** => **FD** | **EMPT**
R3. **FD** => **F FD'**
R4. **FD'** => **FD** | ε
R5. **F** => function **I (OPL) ODL B**
R6. **OPL** => **PL** | **EMPT**
R7. **PL** => **P** | **P , PL**
R8. **P** => **ID : Q**
R9. **Q** => int | boolean | real
R10. **B** => **{ SL }**
R11. **ODL** => **DL** | **EMPT**
R12. **DL** => **D; DL'**
R13. **DL'** => **DL** | ε
R14. **D** => **Q ID**
R15. **ID** => **I** *If* **I** *exist and is from* ***SCAN*** *production, then*
{ gen_instr (STDIN, NIL };
{ gen_instr (POPM, get_address(id)) };
If **I** *exist and is from* ***PRINT*** *production, then*
{ gen_instr (STDOUT, NIL };
ID'
R16. **ID'** => **,ID** | ε
R17. **SL** => **S SL'**
R18. **SL'** => **SL** | ε
R19. **S** => **COM** | **A** | **IF** | **RETURN** | **PRINT** | **SCAN** | **WHILE**
R20. **COM** => **{ SL }**
R21. **A** => **I = E** { gen_instr (POPM, get_address(id)) };
R22. **IF** => if **(C) S IF'**
R23. **IF'** => ifend {back_patch(instr_address)} { gen_instr (STDOUT, NIL };
| else { gen_instr (JUMP, NIL } {back_patch(instr_address)} **S** ifend
R24. **RETURN** => return **RETURN'**
R25. **RETURN'** => ; | **E** ;
R26. **PRINT** => put (**E**);
R27. **SCAN** => get (**ID**);
R28. **WHILE** => while (**C**) **S** whileend
R29. **C** => **E R-op E**
R30. **R-op** => == | ^= | > | < | => | =<
R31. **E** => **TE'**
R32. **E'** => +T { gen_intsr (ADD, nil) } **E'** | - **T** { gen_intsr (SUB, nil) } **E'** | ε
R33. **T** => **FAC T'**
R34. **T'** => ***FAC** { gen_instr (MUL, nil) } **T'** | /**FAC** { gen_instr (MUL, nil) } **T'** | ε
R35. **FAC** => **- PRIM** | **PRIM**
R36. **PRIM** => I { gen_instr (PUSHM, get_address(id) } |
INT { gen_instr (PUSHI, get_address(integer) } |
I { gen_instr (PUSHM, get_address(id) } **(ID)** |
(E) | **REAL** | **true** | **false**
R37. **EMPT** => ε

2. Our program will use a type of top-down parser as Recursive Descent Parser to analyze the production rules.
3. We modified our **Lexer**() to get a text file as input, generate tokens and their corresponding lexeme, and store the results into a vector of type TokenLexPair.
4. Our **Parser()** should print to an output file including the tokens, lexemes and the production rules used:
   - At first, the program will write the token and lexeme found.
   - Then, the program will print out all productions rules used for analyzing this token.
   - A "print statement" at the beginning of each function that will print the production rule.
   - The program contains a "switch" with the "print statement" so that it can turn it on or off.
5. Error handling: if a syntax error occurs, our **Parser()** will generate a meaningful error message, such as token, lexeme, line number, and error type. Then our program may exit due to encountering error syntax.
6. Creating Symbol Table Handling: Every identifier declared in the program should be placed in a symbol table and accessed by the symbol table handling procedures:
   - Each entry in the symbol table should hold the lexeme, and a "memory address" where an identifier is placed within the symbol table.
   - The table includes a procedure that will check to see if a particular identifier is already in the table, a procedure that will insert into the table and a procedure that will printout all identifiers in the table. If an identifier is used without declaring it, then the parser should provide an error message. Also, if an identifier is already in the table and wants to declare it for the second time, then the parser should provide an error message. Also, the program can check the type match.
7. Generating the assembly code:
   - We modified the parser according to the simplified Rat18F as described following:
     - the program has NO <Function Definitions>
     - No "real" type is allowed
     - Consider that "true" has an integer value of 1 and "false" has an integer value of 0.
     - No arithmetic operations are allowed for booleans.
     - The types must match for arithmetic operations (no conversions)
   - We added code to your parser that will produce the assembly code instructions. The instructions should be kept in an array and at the end, the content of the array is printed out to produce the listing of assembly code. Your array should hold at least 1000 assembly instructions. The instruction starts from 1.
   - The listing should include an array index for each entry so that it serves as label to jump to. *The compiler should also produce a listing of all the identifiers.*
8. Our target machine is a virtual machine based on a stack with the following instructions:

PUSHI **{Integer Value}** Pushes the {Integer Value} onto the Top of the Stack (TOS)
PUSHM **{ML - Memory Location**} Pushes the value stored at {ML} onto TOS
POPM **{ML**} Pops the value from the top of the stack and stores it at {ML}
STDOUT Pops the value from TOS and outputs it to the standard output
STDIN Get the value from the standard input and place in onto the TOS
ADD Pop the first two items from stack and push the sum onto the TOS
SUB Pop the first two items from stack and push the difference onto the TOS (Second item - First item)
MUL Pop the first two items from stack and push the product onto the TOS
DIV Pop the first two items from stack and push the result onto the TOS

(Second item / First item and ignore the remainder)

GRT Pops two items from the stack and pushes 1 onto TOS if second item is larger otherwise push 0

LES Pops two items from the stack and pushes 1 onto TOS if the second item is smaller than first item otherwise push 0

EQU Pops two items from the stack and pushes 1 onto TOS if they are equal otherwise push 0

NEQ Pops two items from the stack and pushes 1 onto TOS if they are not equal otherwise push 0

GEQ Pops two items from the stack and pushes 1 onto TOS if second item is larger or equal, otherwise push 0

LEQ Pops two items from the stack and pushes 1 onto TOS if second item is less or equal, otherwise push 0

JUMPZ **{IL - Instruction Location}** Pop the stack and if the value is 0 then jump to {IL}

JUMP **{IL}** Unconditionally jump to {IL}

LABEL Empty Instruction; Provides the instruction location to jump to.

Part IV: Any Limitation

None

Part V: Any shortcomings

None