COL226: Programming Languages
II semester 2022-23

Assignment: The Imperative Language Rational-PL0

For those who did not design the command-line interface in the previous assignment, this is an opportunity to redeem yourself by instead implementing the whole language **Rational-PL0** defined below.

Tokens.

Reserved words. rational, integer, boolean, tt, ff, var, if, then, else, fi, while, do, od procedure, print, read, call.

For convenience some other operations/functions which use letters are also regarded as reserved. See below.

Rational Operators.

Unary. ~, +, `inverse` (reserved)

Binary. .+., .-., .*., ./. (rational division)

Unary Conversions. `make_rat`, `rat`, `showRat`, `showDecimal`, `fromDecimal`, `toDecimal` are all also reserved words.

Integer Operators.

Unary. ~, +

Binary. +, -, *, / (div) , % (mod)

Boolean Operators.

Unary. ! (boolean negation)

Binary. && (andalso) , || (orelse)

Relational Operators. =, <>, <, <=, >, >=

Assignment. :=

Brackets. (,), {, }

Punctuation. ;, ,

Identifiers. (A-Za-z)(A-Za-z0-9)*

Comment. Any string of printable ASCII characters enclosed in (*, *)

EBNF

Program	::=	*Block* .
Block	::=	*DeclarationSeq CommandSeq* .
DeclarationSeq	::=	[*VarDecls*] [*ProcDecls*] .
VarDecls	::=	[*RatVarDecls*] [*IntVarDecls*] [*BoolVarDecls*] .
RatVarDecls	::=	rational *Ident* {, *Ident*}; .
IntVarDecls	::=	integer *Ident* {, *Ident*}; .
BoolVarDecls	::=	boolean *Ident* {, *Ident*}; .
ProcDecls	::=	[*ProcDef* {;*ProcDecls*};] .
ProcDef	::=	procedure *Ident Block* .
CommandSeq	::=	{{*Command*;}} .
Command	::=	*AssignmentCmd* | *CallCmd* | *ReadCmd* | *PrintCmd* | *ConditionalCmd* | *WhileCmd* .
AssignmentCmd	::=	*Ident* := *Expression* .
CallCmd	::=	call *Ident* .
ReadCmd	::=	read(*Ident*) .
PrintCmd	::=	print(*Expression*) .
Expression	::=	*RatExpression* | *IntExpression* | *BoolExpression* .
ConditionalCmd	::=	if *BoolExpression* then *CommandSeq* else *CommandSeq* fi .
WhileCmd	::=	while *BoolExpression* do *CommandSeq* od .

Notes related to Rational-PL0

1. The rational and integer numbers are all as defined in the Big Rational Package that you have already implemented.
2. For efficiency, whenever integers and their operations are well within the `(valOf Int.maxInt)` and `(valOf Int.minInt)` limits you may use the SML-NJ system defined `Int` structure. However, there can be no overflows or underflows in integer operations.
3. The language is statically scoped.
4. All identifiers must be declared before use.
5. All procedure definitions are parameter-less. Hence the call is simply to the procedure. Any parameters to the procedure have to be declared before-hand in some enclosing scope.
6. Brief explanations in parentheses are provided for some of the operators whose meaning may not be obvious to some.
7. No definitions are going to be provided for operators whose semantics is obvious. The binary rational operators are distinguished from their integer counterparts by the pair of dots that enclose the operator.
8. All unary operators and functions are in prefix form and all binary operators are in infix with the usual rules of precedence and associativeity.

What you need to do

1. You have already implemented a Big Rational package in the previous assignment. You have also designed a command-line interface for a rational number calculator.
2. Assume each program written in Rational-PL0 is stored in a text-file `<filename>.rat`
3. Use the Big Rational package to design an **interpreter** in SML-NJ for the language Rational-PL0. *You don't need to design or implement any intermediate language.* Interpret the program in the file `<filename>.rat` directly in SML-NJ.

(a) Design a grammar suitable for scanning and parsing according to your taste. You could use the available tools or design a scanner and parser (for top down or recursive-descent parsing) yourself. Document the grammar according to your scanning and parsing strategies in the file `README.md`.

(b) Your grammar for the expressions in Rational-PL0 should be consistent with and an extension of your grammar for rational expressions (already defined in the previous assignment).

(c) The semantics of the expression language is the usual semantics that we all understand and so has been omitted.

(d) The semantics of the language of commands and blocks is a direct and obvious extension of the semantics of the while language discussed in the notes and in the class. *So there are no surprises!*

Note for all assignments in general

1. Some instructions here may be overridden explicitly in the assignment for specific assignments
2. Upload and submit a single zip file called `<entryno>.zip` where `entryno` is your entry number.
3. You are *not* allowed to change any of the names or types given in the specification/signature. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.
4. The evaluator may use automatic scripts to evaluate the assignments (especially when the number of submissions is large) and penalise you for deviating from the instructions.
5. You may define any new auxiliary functions/predicates if you like in your code besides those mentioned in the specification.
6. Your program should implement the given specifications/signature.
7. You need to think of the *most efficient way* of implementing the various functions given in the specification/signature so that the function results satisfy their definitions and properties.
8. In a large class or in a large assignment, it is not always possible to specify every single design detail and clear each and every doubt. So you are encouraged to also include a `README.txt` or `README.md` file containing
   - all the decisions (they could be design decisions or resolution of ambiguities present in the assignment) that you have taken in order to solve the problem. Whether your decision is "reasonable" will be evaluated by the evaluator of the assignment.
   - a description of which code you have "borrowed" from various sources, along with the identity of the source.
   - all sources that you have consulted in solving the assignment.
   - name changes or signature changes if any, along with a full justification of why that was necessary.
9. The evaluator may look at your source code before evaluating it, you must explain your algorithms in the form of comments, so that the evaluator can understand what you have implemented.
10. Do *not* add any more decorations or functions or user-interfaces in order to impress the evaluator of the program. Nobody is going to be impressed by it.
11. There is a serious penalty for code similarity (similarity goes much deeper than variable names, indentation and line numbering). If it is felt that there is too much similarity in the code between any two persons, then both are going to be penalized equally. So please set permissions on your directories, so that others have no access to your programs.
12. To reduce penalties, a clear section called "**Acknowledgements**" giving a detailed list of what you copied from where or whom may be be included in the `README.txt` or `README.md` file.