Assignment: Byte Code Interpreter

Write a Bytecode interpreter for the following byte-code language.

1 Notational Conventions

1. Symbols belonging to the bytecode lanaguage are in `teletype` font. *There are no symbols in the langauge which use uppercase letters*
2. Meta symbols are in *italics* and always start with an Uppercase letter
3. The definitions are given as production rules of a grammar written in the standard form that you must have studied in any course on programming languages.
4. As is customary wherever convenient the usual symbols are used to denote regular expression operations. You must have seen most of these notations used in the man pages on UNIX-like systems. For example

   { } denotes 0 or more occurrences of anything enclosed in the braces,
   [ ] denotes an optional item (0 or 1 occurrence of the item enclosed in brackets)
   * denotes the Kleene star (0 or more iterations)
   $^+$ denotes the Kleene plus (1 or more iterations)
   | denotes an alternative

5. All words and symbols in the language are reserved and cannot be used as variables in the program
6. `val` denoting *call-by-value* and `ref` denoting *call-by-reference* are the only two parameter passing mechanisms.
7. The notation for pointer referencing/dereferencing use the unary operators `addr` and `deref` and I sincerely hope they are consistent with the slides in the overview presented earlier.
8. The symbol `[]` stands for array indexing and shopuld also be consistent with the slides. Typically in

   $$VarName1\ [\ ]\ VarName2$$

   $VarName1$ is the array name and $VarName2$ is the index. Due to the restriction to 3 operands one cannot make assignments such as `a [] i := b [] j`. Further one cannot mix array-indexing operations with referencing and dereferencing. In each case simple variables *must* be introduced.

9. Note that `--` is the binary subtraction operation, whereas `-` denotes the unary minus. Also see *Integer* and *Float*.
10. `!` stands for boolean negation, `|` stands for boolean disjunction and `&` for boolean conjunction.

2 Language Definition

1. This is a slightly *glorified* assembly language. So the scope rules are such that all new scopes within the global scope are all mutually disjoint.
2. The *global variables* are those that occur in the *Program* statement rule.

3. Procedures do have names, which are simply identifiers. Procedures are delimited by a `begin` $\cdots$ `end`, so their scopes are clearly delimited. Procedure names are also labels, but other labels usually used for pure transfer of control have a certain lexical property defined in the rule for *Label*.
4. *ReadInstr* and *WriteInstr* may be used for terminal i/o. Large arrays will have to be input or output by defining appropriate loops with labels and *GotoInstr* and/or *IfInstr* instructions.
5. Since a procedure might have a large number of formal parameters, individual formal and actual parameters have been separated out into distinct instructions for each parameter. Clearly, (large) arrays cannot be parameters except when the parameter-passing mechanism is `ref`. In such cases the whole array is passed as a reference parameter, so as to avoid duplicating the array.
6. It is obvious that a procedure call is not a single instruction, but would actually be a a sequence of as many *ArgInstr* instructions as there are parameters for the procedure followed by *CallInstr*. Similarly, a procedure declaration, would be immediately followed by a sequence of as many *FormalParamInstr* as necessary, before any other executable instruction of the procedure.
7. There is no recursion anywhere in this language (since this is primarily an assembly language).

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \texttt{program}\ \mathit{ProgramName}\ \mathit{InstrSeq}\ \texttt{exit}\ \{\mathit{ProgramUnit}^*\} \\
\mathit{ProgramUnit} & \rightarrow & [\mathit{ProcedureName}\ ::]\ \texttt{begin}\ \mathit{InstrSeq}\ \texttt{end} \\
\mathit{InstrSeq} & \rightarrow & \{[\mathit{Label}\ :]\mathit{Instr}[\ \#\ \mathit{Comment}]\}^* \\
\mathit{Instr} & \rightarrow & \mathit{FormalParamInstr} \\
 & | & \mathit{AssignInstr} \\
 & | & \mathit{GotoInstr} \\
 & | & \mathit{IfInstr} \\
 & | & \mathit{CallInstr} \\
 & | & \mathit{ArgInstr} \\
 & | & \mathit{ReturnInstr} \\
 & | & \mathit{WriteInstr} \\
 & | & \mathit{ReadInstr} \\
\mathit{FormalParamInstr} & \rightarrow & \texttt{param}\ \mathit{VarName}\ \ \mathit{ParamMech} \\
\mathit{ParamMech} & \rightarrow & \texttt{val}\ |\ \texttt{ref} \\
\mathit{AssignInstr} & \rightarrow & \mathit{VarName} := \mathit{Expression} \\
 & | & \mathit{VarName} := \mathit{VarName}\ [\ ]\ \mathit{Operand} \\
 & | & \mathit{VarName} :=\ \texttt{addr}\ \mathit{VarName} \\
 & | & \mathit{VarName} :=\ \texttt{deref}\ \mathit{VarName} \\
 & | & \texttt{deref}\ \mathit{VarName} := \mathit{Operand} \\
\mathit{GotoInstr} & \rightarrow & \texttt{goto}\ \mathit{Label} \\
\mathit{IfInstr} & \rightarrow & \texttt{if}\ \mathit{RelExpr}\ \texttt{goto}\ \mathit{Label} \\
\mathit{CallInstr} & \rightarrow & \texttt{call}\ \mathit{ProcedureName} \\
\mathit{ArgInstr} & \rightarrow & \texttt{arg}\ \mathit{Operand} \\
\mathit{ReturnInstr} & \rightarrow & \texttt{return} \\
\mathit{WriteInstr} & \rightarrow & \texttt{print}\ \mathit{VarName}\ \{,\ \mathit{VarName}\} \\
\mathit{ReadInstr} & \rightarrow & \texttt{read}\ \mathit{VarName}\ \{,\ \mathit{VarName}\} \\
\mathit{Expression} & \rightarrow & \mathit{Operand}\ \ \mathit{BinaryOpr}\ \ \mathit{Operand} \\
 & | & \mathit{UnaryOpr}\ \ \mathit{Operand} \\
 & | & \mathit{Operand} \\
\mathit{RelExpr} & \rightarrow & \mathit{Operand}\ \ \mathit{RelOpr}\ \ \mathit{Operand} \\
 & | & [\ !]\mathit{Operand} \\
\mathit{Operand} & \rightarrow & \mathit{VarName} \mid \mathit{Const} \\
\mathit{BinOpr} & \rightarrow & \mathit{IntOpr} \mid \mathit{RelOpr} \mid \mathit{BoolOpr} \mid \mathit{FloatOpr} \\
\mathit{IntOpr} & \rightarrow & +\ |\ --\ |\ *\ |\ \texttt{div}\ |\ \texttt{mod} \\
\mathit{FloatOpr} & \rightarrow & +\ |\ --\ |\ *\ |\ / \\
\mathit{BoolOpr} & \rightarrow & \&\ |\ | \\
\mathit{RelOpr} & \rightarrow & =\ |\ !=\ |\ <\ |\ <=\ |\ >\ |\ >= \\
\mathit{UnaryOpr} & \rightarrow & -\ |\ \texttt{addr}\ |\ \texttt{deref} \\
\mathit{Const} & \rightarrow & \mathit{Integer} \mid \mathit{Float} \mid \mathit{Boolean} \\
\mathit{Integer} & \rightarrow & 0 \mid [\ -]\mathit{NZDigitDigit}^* \\
\mathit{Float} & \rightarrow & [\ -]\mathit{Digit}^+\ .\mathit{Digit}^+ \\
\mathit{Boolean} & \rightarrow & \texttt{true}\ |\ \texttt{false} \\
\mathit{Label} & \rightarrow & \texttt{L}\mathit{Integer} \\
\mathit{VarName} & \rightarrow & \mathit{Identifier} \\
\mathit{ProcedureName} & \rightarrow & \mathit{Identifier} \\
\mathit{ProgramName} & \rightarrow & \mathit{Identifier} \\
\mathit{Identifier} & \rightarrow & \mathit{Letter}\{\mathit{Letter} \mid \mathit{Digit} \mid\ \_\}^* \\
\mathit{Letter} & \rightarrow & \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \\
\mathit{Digit} & \rightarrow & \texttt{0} \mid \mathit{NZDigit} \\
\mathit{NZDigit} & \rightarrow & \texttt{1} \mid \ldots \mid \texttt{9}
\end{array}
$$

3 FAQs

This is a collections of FAQ's which came up during the process leading to the submission of the assignment.

Q. What does '::' mean in the program rule of 2nd assignment

A. Just syntactic sugar to distinguish it from a label. Want to get rid of it? Go ahead.

Q. Generally an interpreter will execute the instructions one by one. But since in our grammar, call to procedure always come before definition,we need to keep on building the AST till the end of the input. At the end we can traverse the AST and execute the instructions. This approach is similar to that of compiler where we traverse through the whole code before generating code. The same applies to forward jumps using goto. Will it be acceptable solution that the instructions are executed after the input to interpreter is complete ?

A. That is not completely true. Every assembly language allows for forward jumps, but assembly programs are always interpreted. It only means you may have to scan the entire program once before you actually interpret it instruction by instruction. But you don't necessarily have to construct the AST. You can

1. Simply scan the entire program once, collect all forward references. A variation of this is that you can simply create a symbol table for all identifiers.
2. Resolve each reference as and when you get its definition. Keep this information handy for the for the actual interpretation.
3. After checking that your table has no unresolved references, execute each instruction with the help of this new reference table. otherwise throw out the program with an error message

   `unresolved reference: <identifier> in line <line no.>`

Q. Another minor issue: the grammar allows assignment like -

Variable := addr1 (Address of a constant)

using rules of AssignInstr, Expression, UnaryOpr, Operand, Const.

A. But you will require them anyway since you are going to store constants somewhere and you need access to them.

Q. While passing the arguments as references. We can as well pass a pointer to a local variable with-in the procedure and then this gets referred by the called procedure.Do we need to handle such cases and thus have complete simulation for activation records.?

A. You have actually hit upon one of the important reasons why we SHOULD discourage programming in Assembly and encourage HIGH-LELVEL programming. What you are saying is possible but you don't need to handle it, if you are clear that this is only an intermediate representation which will not be allowed to a user programmer of your HIGH-LEVEL language.

The reason why ASSEMBLY-LEVEL language programming is still being encouraged especially in areas like real-time and embedded systems is due to performance. The fact is that the tradeoff is between STRUCTURE, ABSTRACTION, READABILITY etc on the one hand against (PERCEIVED) REQUIRED performance on the other.

In a first level compiler course it is NOT NECESSARY to handle all this.

Q. Also the type of the variable passing gets determined at the time of formal param declaration ,should it be at arg declaration time?

A. NO. For actual parameters, it just needs checking that it is consistent with the formal parameter. That should be checkable easily from the symbol table that you construct.

Q. And one more thing the grammar allows array indexing by bools and reals i guess we can forget that.

A. Yes, you can safely forget that.

Q. But we do have references as a possible type of data. What operations do we allow on them. I guess addition only.

A. Yes, that is right. The other operations are reading and writing into the locations pointed by them.

Q. Can there be two variables one local and one global of the same name then what's the visibility rule?

A. The usual rule. When control is in the procedure containing the variable, only the local one is visible. Outside it only the global is visible.