Building a Small new Interpreter for Arithmetic Expressions of Pseudo Language by using Polish Notation

Su Myat Thu
University of Computer Studies (Mandalay)
sumyatthu3387@gmail.com

Abstract

Since most imperative languages have similar control structures and operational logic, a pseudo programming language with clearer syntax could be more appropriate as a first teaching language. Therefore, it is an ideal candidate for teaching computation concepts. However, the novices normally would like to try to execute their examples, there's no platform for running pseudo code. Pure interpretation is an easy implementation and a phenomenon that interpreter program executes source machine statements. So, our system intends to make pseudo code execute by using this method in order to help pseudo users see their outputs. We also use polish notation in order to evaluate the arithmetic expressions in a pseudo program.

1. Introduction

Pseudo is a very close to the people and so every beginner in programming language firstly learns this language to train their logic. However, the well-known bottleneck of this language is user can't see their outputs. So, in this system, we try to execute pseudo programs using pure interpretation method. There are three language implementation methods. They are pure interpretation, compilation and hybrid implementation system.

In pure interpretation, programs can be interpreted by another program called an interpreter, with no translation whatever. The interpreter program acts as a software simulation machine whose **fetch-execute cycle** deals with high level language program statements rather than machine instructions. Although the execution is ten to hundred times slower than in compiles systems, this system allows easy implementation of memory source-level debugging operations because all run-time error messages can refer to source-level units. In compilation, programs can be translated to machine language, which can be executed directly on the computer. This method has the advantage of very fast program execution, once the translation process is complete. But, this process is far more complex.

The last, hybrid implementation systems are a compromise between compilation and pure interpretation. Instead of translating intermediate language code to machine code, it simply interprets the intermediate code. Translating high-level language programs to an intermediate language designed to allow easy interpretation is more complex than pure interpretation systems.

For this reasons, pure interpretation system will be the most suitable method in making pseudo program run. Our system will also use lexical and syntax analysis for checking pseudo format input by the user whether it is true or false. This system intends to make pseudo code execute in order to help pseudo users see their outputs.

2. Related works

There has been lot of discussion on choosing the right language for the first programming concepts and building interpreter. Olsen [1] used pseudo code as a design tool in an inductor CS course. Students used pseudo code to define the solution, and then implemented the actual program with C++. Robert E. Filman returned to the question of what distinguishes AOP languages by considering how the interpreters of AOP languages differ from conventional interpreters. ROBERT W.SEBESTA discussed three language implementation methods, pure interpretation, compilation and hybrid-implementation system.

3. Theory background

In this section, background theories are discussed.

3.1 Interpreter

An interpreter normally means a computer program that execute, i.e. performs, instructions written in a programming language. Loosely speaking, an interpreter actually does what the program says to do.

3.2 Lexical analysis

In computer science, lexical analysis is the process of converting a sequence of characters into a sequence of tokens. Programs performing lexical analysis are called lexical analyzers or lexers.

3.2.1 Token. A token is a categorized block of text. The block of text corresponding to the token is known as a lexeme. A lexical analyzer divides a sequence of characters into tokens (tokenization) and categorizes them according to function, giving them meaning. A token can look like anything; it just needs to be a useful part of the structured text.

3.3 Parser

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

3.4 Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. A parser can be constructed for any grammar. Programming language parsers almost always make a single left-to-right scan over the input, looking ahead one token at a time.

3.5 Parse tree

One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define. These hierarchical structures are called the parse tree, and it has the important purpose of making explicit the hierarchical syntactic structure of sentences that is implied by the grammar.

3.6 Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence.

If the grammar is ambiguous, we can disambiguate these grammars by specifying the associability and precedence of the arithmetic operators. Suppose we wish to give the operators the following precedence in decreasing order:

- (Unary minus)
↑
* /
+ -

3.7 Polish notation

Polish notation was described in the 1920s by Polish mathematician Jan Lukasiewicz as a logical system for the specification of mathematical equations without parentheses. There are two versions, prefix notation and postfix notation. In prefix notation, the operators are placed before the operand. In postfix notation, this order is reversed. Several conventions exist for the evaluation of arithmetic expressions. Prefix notation is known as Polish Notation after the nationality of Lukasiewicz. Similarly, postfix notation is known as Reverse Polish Notation (RPN).

In *infix notation*, operators appear between operands:

A+B*C

In *prefix notation*, operators precede operands:

+A*BC

In *postfix notation*, operators follow operands:

ABC*+

4. Design of the system

This section explains the design of the system.

4.1 Flow design of the system

In Figure 1, firstly, lexical analysis is the process of converting a sequence of characters for input pseudo program into a sequence of tokens. The resultant tokens are used by syntax analysis. We use this second process for determining if a string of tokens can be generated by a grammar. We can determine whether the input program format is true or false by using it. So, this analysis doesn't intend to produce abstract syntax tree (AST) in our system.

Last, if the input format is true, we will interpret it for producing result. In this process, we mainly use polish notation for evaluating arithmetic expressions of pseudo program.

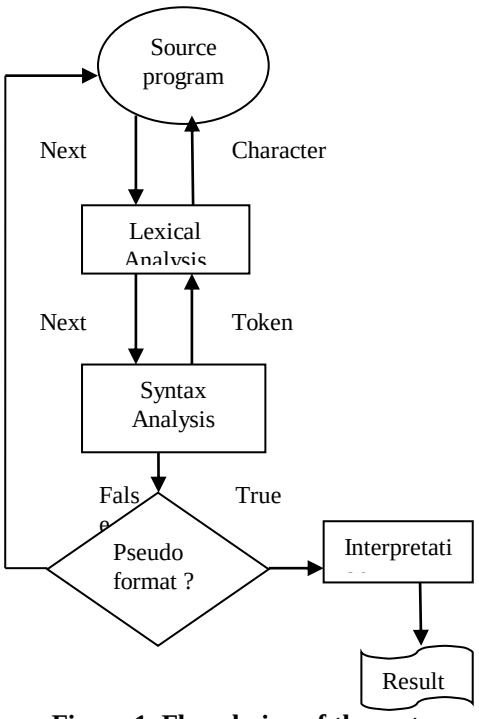

Figure 1. Flow design of the system

The computer usually evaluates an arithmetic expression written in infix notation in two steps. First it converts the expression to postfix notation, and then it evaluates the postfix expression. In each step, the stack is the main tool that is used to accomplish the given task. We illustrate these applications of stacks in reverse order. That is, first we show how stacks are used to transform infix expression into postfix expressions and then we show how stacks are used to evaluate postfix expressions.

4.2 Transforming infix into postfix expressions algorithm

We suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
    a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than⊗ .
    b) Add ⊗ to STACK.

[End of If structure.]

6. If a right parenthesis is encountered, then:
    a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
    b) Remove the left parenthesis. [Do not add the left parenthesis to P.]

    [End of If structure.]

[End of Step 2 loop.]

7. Exit.

Continuously, we use the following algorithm for finding the VALUE of an arithmetic expression P written in postfix notation.

4.3 Evaluation of a postfix expression algorithm

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator ⊗ is encountered, then:
    a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
    b) Evaluate B ⊗A.
    c) Place the result of (b) back on STACK.

    [End of If structure.]

[End of If structure.]

5. Set VALUE equal to the top element on STACK.
6. Exit.

5. Implementation of the system

In this section, we illustrate our system using the following pseudo example program.

```
BEGIN
    a=10;
    b=20;
    c=2;
    Display a+ b*c;
END
```

5.1 Using lexical analysis in the system

Lexical analysis breaks up input program into the following tokens. In our pseudo example program,

Table 1. Generating tokens

Input	Token	Lexeme

BEGIN
a=10;
b=20;
c=2;
Display a+
b*c;
END
	Keyword	BEGIN

	Identifier	a
	Operator	=
	Number	10
	Punctuator	;
	Identifier	b
	Operator	=
	Number	20
	Punctuator	;
	Identifier	c
	Operator	=
	Number	2
	Punctuator	;
	Keyword	Display
	Identifier	a
	Operator	+
	Identifier	b
	Operator	*
	Identifier	c
	Punctuator	;
	Keyword	END

where lexemes, BEGIN, Display and END indicate the keywords, a, b and c indicate the identifiers, ; indicates the punctuator, 10, 20 and 2 indicate integer numbers and =, + and * are operators. See in **Table 1**.

Each time the parser needs a token, it sends a request to the lexical analyzer. Then, the lexical analyzer reads as many characters from the input stream as it is necessary to construct a single token. For instance, an *integer* token may contain any sequence of numerical digit characters. The lexical analyzer may report an error during scanning (e.g., when it finds an end-of-file in the middle of a string). Otherwise, when a single token is formed, the lexical analyzer is suspended and returns the token to the parser. See the figure1. The parser will repeatedly call the lexical analyzer to read all the tokens from the input stream or until an error is detected (such as a syntax error).

5.2 Using syntax analysis in the system

We use syntax analysis for checking the input pseudo program whether it is true or false pseudo format. In checking process, we must apply following grammar rules.

5.3 Grammar rules for pseudo example program

1. <program> → BEGIN <st-list> END
2. <st-list> →<st> <st-list> ′
3. <st-list>′ →<st><st-list> ′
4. <st-list>′ →Є
5. <st> → <assign>
6. <st> →<if>
7. <st> → <for>
8. <assign> →id= <exp>;
9. <exp> → <term> <exp>′
10. <exp> ′ →+ <term> <exp>′
11. <exp>′ → - <term> <exp>′
12. <exp>′ →Є
13. <term> →<factor> <term>′
14. <term> ′ →*<factor><term>′
15. <term>′ → /<factor><term>′
16. <term>′ →Є
17. <factor> → id
18. <factor> →num
19. <factor> →(<exp>)
20. <display> → Display <display-list>;
21. display-list> → <exp>
22. <if> →if <exp> then <st-list> end if
23. <for> →for <exp> from <exp> to <exp> do <st-list> enddo

5.4 Parse tree for example program

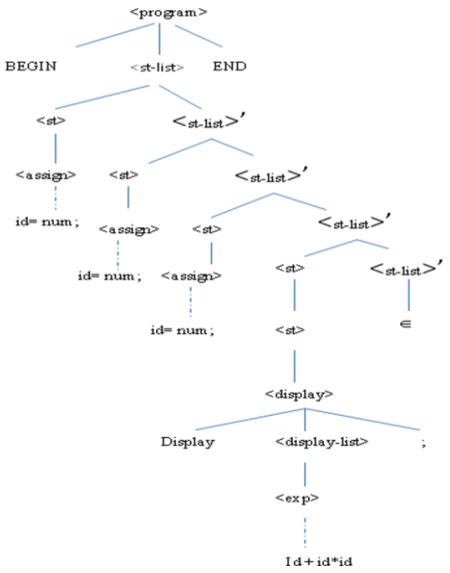

Figure 2. Parse tree for example program

5.5 Transforming infix into postfix expressions

Let Q be our infix expressions:

Q: a+ b*c

First we push "(" onto STACK, and then we add ")" to end of Q to obtain:

Q: a + b * c)

(1) (2) (3) (4) (5) (6)

The elements of Q have now been labeled from left to right for easy reference. **Table 2** shows the

status of STACK and of the string P as each element of Q is scanned.

After step 6 is executed, the STACK is empty and

P: a b c* +

which is the required postfix equivalent of Q.

Table 2. Status of STACK for infix into postfix expression

Symbol Scanned	STACK	Expression
(1) a	(	a
(2) +	(+	a
(3) b	(+	a b
(4) *	(+*	a b
(5) c		a b c
(6))		a b c * +

5.6 Evaluation of postfix expression

Expression P written in postfix notation for example Pseudo program:

P: 10, 20, 2, *, +

(Commas are used to separate the elements of P so that 20, 10 is not interpreted as the number 20102.) The equivalent infix expression Q follows:

Q: 10+20*2

Note that parentheses are necessary for the infix expression Q but not for the postfix expression P. We evaluate P by simulating **Algorithm 3.3.** First we add a sentinel right parenthesis at the end of P to obtain

P: 10, 20, 2, * +)

(1) (2) (3) (4) (5) (6)

The elements of P have been labeled from left to right for easy reference. **Table 3** shows the contents of STACK as each element of P is scanned. The final number in STACK, 10, which is assigned to VALUE when the right parenthesis ")" is scanned, is the value of P.

Table 3. Evaluating the postfix expression

Symbol Scanned	STACK
(1) 10	10
(2) 20	10,20
(3) 2	10,20,2
(4) *	10,40
(5) +	50
(6))	

6. Conclusion

Programming environments have become important parts of software development systems, in which the language is just one of the components. The major methods of implementing programming languages are compilation, pure interpretation, and hybrid implementation. Pure interpretation is an easy implementation of memory source-level debugging operations, because all run-time error messages can refer to source-level units. Our system intends to pseudo user monitor their program's outputs applying pure interpretation method. Before we interpret input program, we check it whether it is true or false pseudo format by using lexical and syntax analysis.

7. Limitations and further extensions

This system can only interpret the correct syntax format of the **Pseudo** program. So, we hope that some of the students will try to interpret the input **Pseudo** by constructing automatic correction for this program even if it is not correct syntax. And this system can only interpret the keywords (if, else, for, BEGIN, END, Display). Other students will try to interpret all keywords available in pseudo format.

8. References

[1] Olsen, A.L., "Using Pseudo code to Teach Problem Solving", Journal of Computing Sciences in Colleges, 21(2):231-236, 2005.

[2] Robert E. Filman, "Understanding AOP through the Study of Interpreters", Research Institute for Advanced Computer Science, NASA Ames Research Center, MS 269-2, Moffett Field, CA 94035.

[3] ROBERT W.SEBESTA, "Concepts of Programming Language", ISBN 81-7808-161-X, Fourth Edition, 2001, Addison Wesley Longman (Singapore) Pte.Ltd., Indian Branch, 482 F.I.E. Patparganj, Delhi 110092, India.

[4] P.M.LEWIS II, D.J.ROSENKRANTZ, R.E.STEARNS, "Compiler Design Theory", ISBN 0-201-14455-7, DEFGHIJKL - XA - 89876543210.

[5] Seymour, Lipschutz, Ph.D. "Theory and Problems of Data Structures", Professor of Mathematics, Temple University.

[6] Elliot Berk, "JLex: A Lexical analyzer generator for Java ™", Department of Computer Science, Princeton University, Version 1.2, May 5, 2007.