Samuli Lehtonen

Mini-PL Interpreter Documentation

Token patterns as regular expressions

Token name in the interpreter	**Meaning**	**Regular expression**
INTEGER	Represents integer value. For example: 52 or 12	(?<=\s|^)\d+(?=\s|$)
PLUS	Represents the ‘+’ symbol	(+)
MINUS	Represents the ‘-‘ symbol	(-)
MUL	Represents the ‘*‘ multiplication operator	(*)
DIV	Represents the ‘/‘ division operator	(/)
PARENLEFT	Represents the ‘(‘ symbol	([(])
PARENRIGHT	Represents the ‘)‘ symbol	([)])
EOF	Represents the end of file	\Z
ASSIGN	Represents the “:=” assignment symbol	(:=)
SEMICOLON	Represents the ‘;’ semicolon that ends a statement	(;)
IDTOKEN	Represents a variable name	(?<!\S)(?!var|int|string|bool|print|read|assert|for|in|do|end)[A-Za-z][\w$]*(\.[\w$]+)?(\[\d+])?(?!\S) **NOTE: In the lex analyser the matching is done by first reading the token and then checking if the keyword is defined in a hash map.**
TWODOT	Represents the ‘:’ symbol	(:)
VAR	Represents the “var” keyword	(?<!\S)var(?!\S)
INTEGER_TYPE	Represents the “int” type specifier word	(?<!\S)int(?!\S)

STRING_TYPE	Represents the “string” type specifier word	(?<!\S)string(?!\S)
BOOL_TYPE	Represents the “bool” type specifier word	(?<!\S)bool(?!\S)
STRING	Represents a string value. For example “abcd\n”	"([^"\]|(\[\tn"])*"
PRINT	Keyword for printing out variables	(?<!\S)print(?!\S)
READ	Keyword for reading user input.	(?<!\S)read(?!\S)
ASSERT	Keyword for telling whether a statement is true	(?<!\S)assert(?!\S)
EQUAL	Represents the ‘=’ symbol for checking equality	(=)
FOR	Keyword for starting a for loop and ending it	(?<!\S)for(?!\S)
IN	Keyword used in for loop variable evaluation	(?<!\S)in(?!\S)
DO	Keyword used in for loop	(?<!\S)do(?!\S)
END	Keyword used at the end of for loop before the “for”	(?<!\S)end(?!\S)
TWO_CONCECUTIVE_DOTS	Represents the “..” in for loop	([.][.])
LESS_THAN	Represents the ‘<’ operator	(<)
LOGICAL_AND	Represents the ’&’ operator	(&)
LOGICAL_NOT	Represents the ’!’ operator	(!)

Non-token patterns include: multiline comments (/* and */), single line comments (//) and whitespaces such as \n and \t. Some of the regex expressions could also be combined but I put them individually for clarity.

Usage of the interpreter

When you launch the interpreter, it will ask you to provide a name of the file (include the end like .txt) where the code is and the file must be inside the same folder as the interpreter .exe file. If the file exists, then the interpreter will process it and if it has no errors, then it will be executed normally. You can build the interpreter by opening the .sln solution file with Visual Studio and just clicking run or pressing F5. I used Visual Studio 2017 and Windows 10. I haven't tried to open the project with other versions. **You can find the pre-built MiniPL_Interpreter.exe in my** google drive **because the attachment is too big to send as it contains the .NET dll files that the department computers don't have.** It should work on 64-bit Windows 7 and higher. I tested the pre-built version on the CS department computer with Windows and it worked good. When you extract the zip file from my google drive**, you will find the MiniPL_Interpreter.exe inside the publish folder.**

Google drive link for the pre-built interpreter:
https://drive.google.com/open?id=1bpqK9RrfIxj85BxwtbaPTdxprDsfgc1S

The architecture of the interpreter

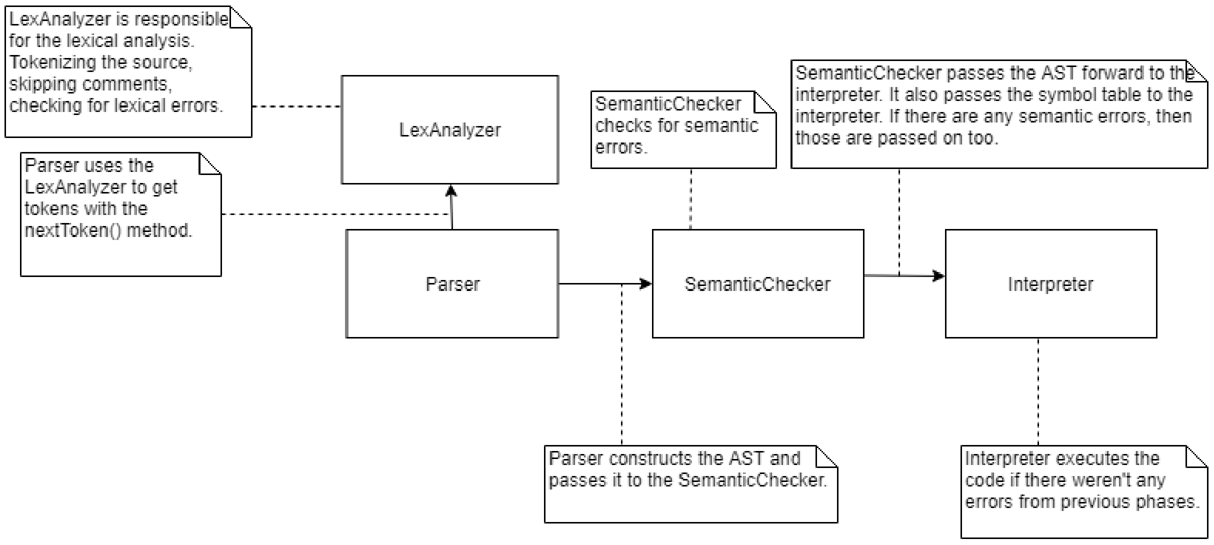
LexAnalyzer is responsible for the lexical analysis. Tokenizing the source, skipping comments, checking for lexical errors.
Parser uses the LexAnalyzer to get tokens with the nextToken() method.
LexAnalyzer
Parser
SemanticChecker checks for semantic errors.
SemanticChecker passes the AST forward to the interpreter. It also passes the symbol table to the interpreter. If there are any semantic errors, then those are passed on too.
SemanticChecker
Interpreter
Parser constructs the AST and passes it to the SemanticChecker.
Interpreter executes the code if there weren't any errors from previous phases.

Figure 1: The architecture of the interpreter

The figure 1 displays the architecture of the interpreter. The process starts from the lexical analysis, but in fact the parser calls the lex analyser to pass the next token when a new token is needed. This is done until the end of file (EOF) token is reached. The parser will continuously build the AST and looks for syntactic errors. When the AST is ready, it is passed to the semantic checker which checks for semantic errors and binds names to their declarations. After the semantic analysis is done, the AST, symbol table and possible errors are passed on to the interpreter. If there are errors, then those are displayed the interpretation process stops. If there no errors, then the program will be executed.

The Interpreter and the SemanticChecker derive from the NodeInterpreter class which has one method *VisitNode(AST node).* This method uses reflection to determine the name of the node's class and it constructs the method name based on that. For example, if we pass a ForAST to the method, then the name of the method that will be called is VisitForAST. Now the interpreter and the semantic checker can derive the NodeInterpreter base class and just implement methods for every AST type like VisitForAST, VisitStatementListAST, VisitNumericAST etc. This is basically an improved version of the node visitor pattern.

Structure of abstract syntax trees (AST)

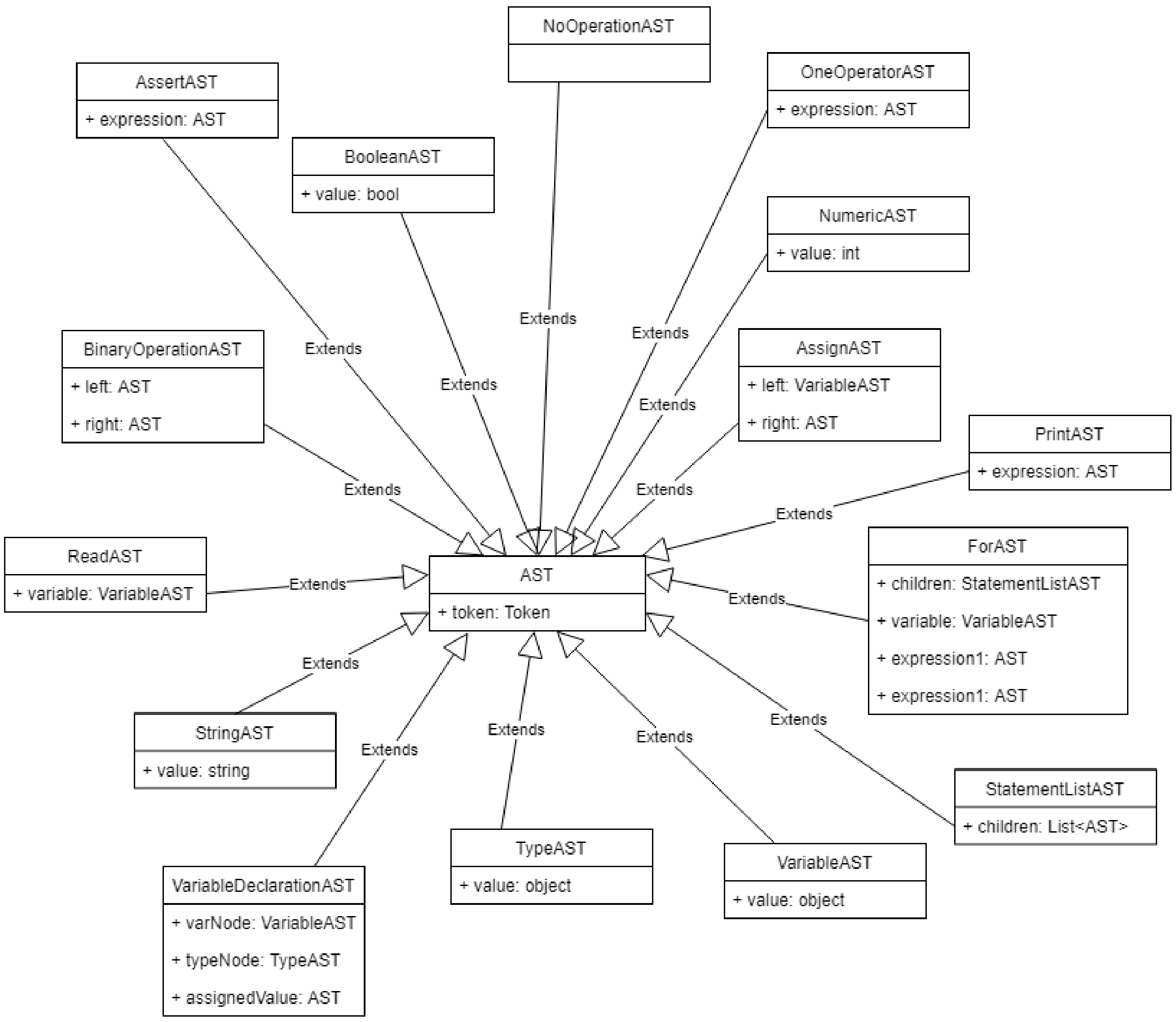

Figure 2: Class diagram - Every AST node is derived from the AST base class

As can be seen from the figure 2, every AST node is derived from the base class AST which has one member variable "token" The token represents the token of the node and because every AST node is derived from the base class AST, they also have the token variable.

We can also notice that many of the AST nodes have other AST nodes as their class variables. For instance, the BinaryOperationAST has two other ASTs, the left and the right side of the equation. The whole program itself is one StatementListAST which contains all the statements in the program.

Modified Mini-PL LL (1) grammar

```
<prog>                  ::= <stmt-list>

<stmt-list>             ::= <stmt> “;” <stmt-list-end>

<stmt-list-end>         ::= <stmt-list>

                        | ε

<stmt>                  ::= “var” <var-ident> “:” <type> <optional-assignment>

                        | <var-ident> “:=” <expr>

                        | “for” <var-ident> “in” <expr> “..” <expr>

                        “do” <stmt-list> “end” “for”

                        | “read” <var-ident>

                        | “print” <expr>

                        | “assert” “(“ <expr> “)”

<optional-assignment>   ::= “:=” <expr>

                        | ε

<expr>                  ::= <opnd> <expr-end>

                        | <unary-op> <opnd>

<expr-end>              ::= <op> <opnd>

                        | ε

<opnd>                  ::= <int>

                        | <string>

                        | <var-ident>

                        | “(“ <expr> “)”

<type>                  ::= “int” | “string” | “bool”

<var-ident>             ::= <ident>
```

For the original grammar, please refer to the original Mini-PL documents at the end of this document.

Error handling

Lexical errors are detected and reported by the lexer. When lexer arrives at an error situation, the lexing process is immediately stopped and the error is reported to the user. Parser looks for syntactic errors and if an invalid token pattern is detected, then the parsing is stopped immediately and the error is shown to the user. Basically, the whole base of the program is wrapped in a try-catch statement so the execution goes to the catch statement when the lexer, parser or the interpreter throws an exception.

The error handling in the semantic analyser is done by adding all detected errors into a list so the interpreter can check whether any error exists in the list and if the list is empty, then it can proceed to interpret the program. If there are errors, then the errors are printed for the user to see but the interpretation won't proceed.

The error handling on the interpreter focuses on the errors that are possible from the user input because the semantic analyser has already analysed the semantic correctness of the program. For instance, if the code has "var a : int; read x" and the user inputs a string value, then an error will be shown to the user and the interpretation process will be stopped. Also, integer overflow is checked and if too large or small integer is given, then the interpretation is stopped. Division is by zero is also considered. Integer overflow and division by zero are both classified as runtime errors by my interpreter and thus they are reported by the interpreter itself, not the semantic analyser.

There is an error class that represents an error. There are also more specific subclasses of the error class that represent a certain type of error. For example, there is a RuntimeError and LexicalError classes that are used in the respective parts of the interpreter.

Special notes

Because there are no values like "true" or "false" defined in the Mini-PL language syntax, the only way to assign variable to a bool type variable is to assign it by using an expression that evaluates to true. For example, `var a : bool := 1=1;` would assign value true to the variable a.

Because it's not explicitly defined what the default variables are, I chose that the default assignment are the following. String will be assigned an empty string if no initial value is provided, int will be set to zero if no value is provided and bool to false.

For example:

```
var a : string; // Would become a = ""

var b : int; // Would become b = 0

var c : bool; // Would become c = false
```

The '<' operator is also defined for string according to the Mini-PL documentation, but it's not explicitly said how it operated so I decided that it will compared the lengths of two strings and return true whether the first string is shorter than the second string.

It wasn’t explicitly mentioned what special characters should be allowed for strings so I chose the following. For strings, special characters that are allowed in my interpreter are: **\n, \t, \\, \r, \v and \”**

Error will be generated if you try to put any other escape character.

It should also be noted that variable declarations are not allowed inside for loops under my implementation because according to the specification, variables can’t be declared more than once. Of course, there exists a situation where the loop only runs once, but I found it clearer to just disallow the variable declarations inside for loops. My semantic analyser generates a special error if there is a variable declaration inside a loop.

String literals can extend to multiple lines as this wasn’t specifically defined in the Mini-PL syntax document.

Test data

The testing of the program was done by user testing and conducted by myself. All of these example programs were ran through the interpreter and the results were written in this document. I found everything working according to my specifications and noticed a few bugs that I fixed. I tried to make the test data very extensive and cover different kinds of edge cases that I made up.

Trying to reassign the loop control variable inside a for loop

```
var a : int := 5;
for a in 0..5 do
        a := 2;
end for;
```

Result: ERROR [Line 3, Column 4] Trying to assign loop control variable inside a loop

Trying to reassign the loop control variable inside a nested for loop

```
var a : int := 5;
var b: int := 2;
for a in 0..5 do
        for b in 0..4 do
                a := 2;
        end for;
end for;
```

Result: ERROR [Line 5, Column 5] Trying to assign loop control variable inside a loop

Trying to assign string for int variable

```
var a : int := "test";
```

Result: ERROR [Line 1, Column 21] Type mismatch

String concatenation

```
var a : string := "hello ";
var b : string := a + "world!";
print b;
```

Result: hello world!

Fibonacci sequence

```
var a : int := 0;
var b : int := 1;

var times : int;
print "how many numbers to calculate: ";
read times;
var control : int;
for control in 1..times do
        print a;
        print "\n";
        a := a + b;
        b := a - b;
end for;
```

Result: Input 5, result 0,1,1,2,3

Wrong type test

```
var a : intasdasd := 5;
```

Result: PARSE ERROR [Line 1, Column 18] Invalid token sequence

Less than operator test

```
var a : int := 5;
var b : string := "hello";

assert (a < 3);
assert (b < "hellohello");
```

Result: Assertion failed

Print NxN matrix

```
var matrixSize : int;
print "Give matrix size(NxN): ";
read matrixSize;

var c1 : int;
var c2 : int;

for c1 in 1..matrixSize do
        for c2 in 1..matrixSize do
                print c2;
                print " ";
        end for;
        print "\n";
end for;
```

Result: Input: 5, result

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Unclosed comment

```
var a : string := "test";
var b : int := 666;
```

```
/*asdasdasasdasd
```

Result: LEXICAL ERROR [Line 3, Column 18] Unclosed comment

Boolean things

```
var a : bool := 1=1;
var b : bool := 1=2;
assert (a=b);
```

Result: Assertion failed

Logical and (&) operator

```
var a : bool := 1=1;
var b : bool := 1=2;
assert (a&b);
```

Result: Assertion failed

Mini-PL test programs from the documentation

```
var nTimes : int := 0;
print "How many times?";
read nTimes;
var x : int;
for x in 0..nTimes-1 do
print x;
print " : Hello, World!\n";
end for;
assert (x = nTimes);
```

Result: Input 5, result

```
0 : Hello, World!
1 : Hello, World!
2 : Hello, World!
3 : Hello, World!
4 : Hello, World!
Assertion failed
```

```
print "Give a number";
```

```
var n : int;
read n;
var v : int := 1;
var i : int;
for i in 1..n do
v := v * i;
end for;
print "The result is: ";
print v;
```

Result: Input 5, result 120

Testing integer overflow

```
var a : int;
read a;
```

Result: input 55555555555555555555555555555555555555555555555555555555555, result RUNTIME ERROR [Line 2, Column 8] Integer overflow

Testing assigning string to integer variable via read command

```
var a : int;
read a;
```

Result: input “abc”, result RUNTIME ERROR [Line 2, Column 8] Tried to assign non-integer value to integer

String concatenation

```
print "hello " + "\"samuli\"";
```

Result: hello “samuli”

Trying to add integer and string together

```
var a : int := 5+"aa";
```

Result: ERROR [Line 1, Column 18] Type mismatch

Trying to define variable when it’s already defined

```
var a : int := 1;
var a : int := 5;
```

Result: ERROR [Line 2, Column 7] Variable a already declared

Trying to use the logical not operator on int

```
var a : int := !5;
```

Result: ERROR [Line 1, Column 17] Unsupported operation for type int

Trying to use a variable that is not defined

```
print a;
a := 5;
```

Result:

```
ERROR [Line 1, Column 8] Variable not declared
ERROR [Line 2, Column 3] Variable not declared
```

Trying to declare a variable inside a for loop

```
var a : int;
for a in 0..2 do
        var b : int;
end for;
```

Result: ERROR [Line 3, Column 15] Variable declaration inside for loop

Trying to use unsupported escape character

```
var a : string := "asad\p";
```

Result: LEXICAL ERROR [Line 1, Column 24] Invalid character in string after escape character

Trying to use string that is never ended

```
var a : string := "\\;
```

Result: LEXICAL ERROR [Line 1, Column 23] Unclosed string

Trying to divide by constant zero

```
var a : int := 5/0;
```

Result: RUNTIME ERROR [Line 1, Column 18] Attempted to divide by zero

Trying to divide by zero input value

```
var a : int := 5/5;
read a;
print 4/a;
```

Result: Input 0, result RUNTIME ERROR [Line 3, Column 10] Attempted to divide by zero

Testing all the operations

```
var a : int := 5+7-0;
var b : int := 5*2;
var c : int := 10/5;
assert(a=12);
assert(b=10);
assert(c=2);

var d : string := "hello\nworld";
var e : string := d + " hello\nworld";
assert(e="hello\nworld hello\nworld");

assert(a<b);

var f : bool := 1=1;
var g : bool := 1=2;

assert(f=!g);
assert(f&g);
```

Result:

```
Assertion failed
Assertion failed
```