Mini-PL interpreter documentation

Suvi Sipilä, 014220326

Mariehamn March 16, 2020

UNIVERSITY OF HELSINKI
Computer Science Master's programme
CSM14204 - Compilers

Contents

1 Instruction

Give the Mini-PL program code to the CodeExample.txt file. Execute the code at the root of the project by writing ***dotnet run*** in the terminal.

2 Architecture

The program reads the file containing the Mini-PL program code. The program delivers the text to Lexer, which reads and analyzes all the given characters and converts them into a token stream. Lexer uses lists defined by the ListSetting class. The ListSetting class sets lists that contain reserved keywords, character escapes, and token types. The Parser class gets Lexer as its parameter and analyzes the entire nested structure of the code and determines if the given token sequence is valid according to the grammar. The SemanticAnalyzer class gets Parser as its parameter and is responsible for visiting all the AST nodes (figure 2) with the help of NodeVisitor. It also uses SymbolTable to determine whether the types are correct, all variables are defined, and that the code itself makes sense. SymbolTable uses two classes, BuiltinTypeSymbol and VariableSymbol, which extends the abstract Symbol class. BuiltinTypeSymbol is responsible for built-in types such as *int*, *string* and *bool*. VariableSymbol class is responsible for variable symbols like *name = 'x', type = 'int'*. Interpreter class gets SemanticAnalyzer as its parameter, and it is responsible for visiting all the AST nodes with the help of NodeVisitor. Interpreter maintains the *GLOBAL_MEMORY* list that includes run-time values for the variables, like *variable = 'x', value = 1*. If one of the so-called "main classes" (Lexer, Parser, SemanticAnalyzer or Interpreter) detects an error in the code, they throw the error with their Error class and the program stops running.

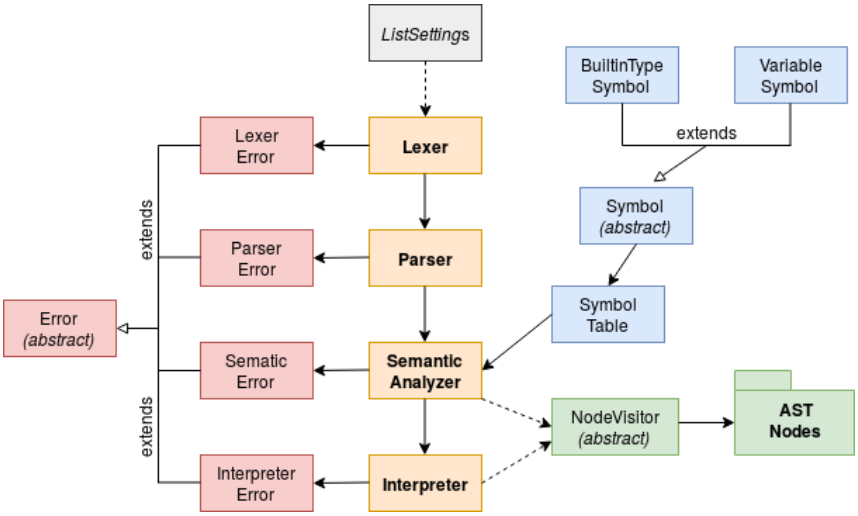

Figure 1: Architecture

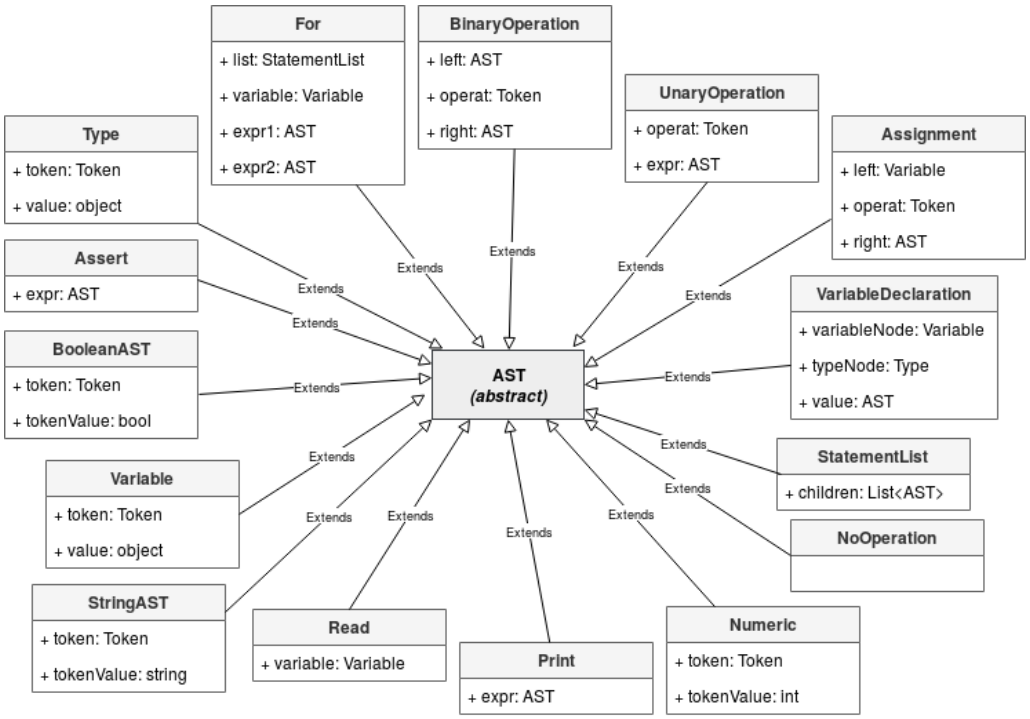

Figure 2: AST nodes

3 Context-free grammmar

Modified context-free grammar for Mini-PL. Keywords are written in lowercase and token types are capitalized.

```
<program>        ::= <statement_list>
<statement_list> ::= ( <statement> ";" )*
<statement>      ::= "var" "ID" ":" <type_spec>
                   | "var" "ID" ":" <type_spec> ":=" <expr>
                   | "ID" ":=" <expr>
                   | "for" "ID" "in" <expr> ".." <expr> "do"
                     <statement_list> "end" "for"
                   | "read" "ID"
                   | "print" <expr>
                   | "assert" "(" <expr> ")"
                   | ε
<expr>           ::= <term> ( ("PLUS" | "MINUS" | "EQUAL"
                   | "LESS" | "AND" | "NOT") <term> )*
<term>           ::= <factor> ( ("MUL" | "DIV") <factor> )*
<factor>         ::= ("PLUS" | "MINUS") <factor> | "INTEGER"
                   | "STRING" | "BOOL" | "(" <expr> ")" | "ID"
<type_spec>      ::= "int" | "string" | "bool"
```

4 Regular expressions

Token type	Regex	Explanation
Operators		
MINUS	-	Integer subtraction
PLUS	+	Integer addition and string concatenation
MUL	*	Integer multiplication
DIV	/	Integer division
EQUAL	=	Equality comparison
LESS	<	Less-than comparison
AND	&	Logical and
NOT	!	Logical not
Keywords		
VAR	var	Variable assigment
FOR	for	For-loop begins
END	end	End of for-loop
IN	in	Defines the integer range of the for-loop
DO	do	Starts the code block of the for-loop
READ	read	Reads integer value or string word from the input stream
PRINT	print	Write out integer or string values
INT	int	Integer type
STRING	string	String type
BOOL	bool	Bool type
ASSERT	assert	Verify assertions
Marks		
LEFTBRACKET	(	Start of nested expression
RIGHTBRACKET	)	End of nested expression
ASSIGN	:=	Assigment
SEMI	;	End of statement
COLON	:	Variable type assigment
DOT	.	Two dots creates an integer sequence
Other		
INTEGER	[0-9]+	Integer value
ID	[a-zA-Z0-9]+	Identifier
EOF		End of input

Figure 3: Mini-PL regular expressions

5 Error handling

Errors are handled in Lexer, Parser, SemanticAnalyzer, and Interpreter. If one of the classes notices an error, they will throw an error message and the program will stop running. The error message contains a description of the error, and in some cases it displays the line and the column number where the error was found. Below are examples of errors and how to handle them.

```
var a : string := "hello;
print a;

>> Unhandled exception.
>> LEXER ERROR
>> End of input while scanning a string literal
```

```
var a := 2;
print a;

>> Unhandled exception.
>> PARSER ERROR
>> Unexpected token -> Token(ASSIGN, :=, position=1:7)
```

```
var a : int := b;
print a;

>> Unhandled exception.
>> SEMANTIC ERROR
>> Identifier not found -> Token(ID, b, position=1:16)
```

```
var a : int := 3;
print a;
var a : string := "Hello";
print a;

>> Unhandled exception.
>> SEMANTIC ERROR
>> Duplicate identifier found -> Token(ID, a, position=3:5)
```

```
var a : int := 3;
print a;
b := "Hello";
print b;

>> Unhandled exception.
>> SEMANTIC ERROR
>> Variable name not found -> Token(ID, b, position=3:1)
```

```
var number : int := 0;
print "Give a number ";
read number;

>> Give a number asd
>> Unhandled exception.
>> INTERPRETER ERROR
>> Invalid input. Expected input value is a number
```

6 Test data

```
// Sample programs
var X : int := 4 + (6 * 2);
print X;

>> 16
```

```
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);

>> How many times? 2
>> 0 : Hello, World!
>> 1 : Hello, World!
>> Assertion fails: FALSE
```

```
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;
print("\n");

>> Give a number 3
>> The result is: 6
```

```
// string concatenation
var hello : string := "Hello,";
var me : string := " is it me ";
var you : string := "you're looking for?\n";
var text : string := hello + me + you;
print text;

>> Hello, is it me you're looking for?
```

```
// assert for string
assert (hello = me);
assert (me < you);

>> Assertion fails: FALSE
>> TRUE
```

```
// assert for bool
var s : int := 2;
var u : int := 3;
assert ((s = 2) ! (u = 2));
assert ((s = 2) & (u = 3));

>> TRUE
>> TRUE
```

```
// math
print 7 + 3 * (10 / (12 / (3 + 1) - 1)) / (2 + 3) - 5 - 3 + (8);
var a : int := 88 / 4;
var b : int := 2;
var c : int := 5;
print a + b + c / (a * b) + (c / 2) + b;

>> 1028
```

```
print "Tell me something: ";
var something : string;
read something;
print "You told: " + something;

>> Tell me something: Hello World!
>> You told: Hello World!
```

7 Work hour log

Date	Time	Work done
17.02.2020	4h	Learning C# and building a project
18.02.2020	6h	Learning C#, start of AST-nodes, easy calculations
19.02.2020	6h	Visitors, multi-digit integers, arithmetic expressions and validation
24.02.2020	6h	Editing visitors and syntax
25.02.2020	6h	String processing done, print done, validation, start of bool-type
26.02.2020	6h	Bool compelete, for- and assert functions. Operations less, not, and, equal and functionality. Refactoring VisitBinaryOperation
02.03.2020	6h	Error-classes, StringBinaryOperations and BoolBinaryOperations done, comment lines
03.03.2020	3h	Writing Mini-PL code to txt-file and testing
04.03.2020	4h	Documentation, architecture picture, regex, contex-free grammar
10.03.2020	4h	Documentation, testing
11.03.2020	5h	Documentation, testing
16.03.2020	4h	Documentation, testing, fixed Lexer's line and column calculations
TOTAL HOURS: 60h		

Figure 4: Work log

A Syntax and semantics of Mini-PL

Syntax and semantics of Mini-PL (27.01.2020)

Mini-PL is a simple programming language designed for pedagogic purposes. The language is purposely small and is not actually meant for any real programming. Mini-PL contains few statements, arithmetic expressions, and some IO primitives. The language uses static typing and has three built-in types representing primitive values: **int**, **string**, and **bool**. The BNF-style syntax of Mini-PL is given below, and the following paragraphs informally describe the semantics of the language.

Mini-PL uses a single global scope for all different kinds of names. All variables must be declared before use, and each identifier may be declared once only. If not explicitly initialized, variables are assigned an appropriate default value.

The Mini-PL **read** statement can read either an integer value or a single *word* (string) from the input stream. Both types of items are whitespace-limited (by blanks, newlines, etc). Likewise, the **print** statement can write out either integers or string values. A Mini-PL program uses default input and output channels defined by its environment. Additionally, Mini-PL includes an **assert** statement that can be used to verify assertions (assumptions) about the state of the program. An **assert** statement takes a **bool** argument. If an assertion fails (the argument is *false*) the system prints out a diagnostic message.

The arithmetic operator symbols '+', '-', '*','/' represent the following functions:

```
"+" : (int, int) -> int            // integer addition
"-" : (int, int) -> int            // integer subtraction
"*" : (int, int) -> int            // integer multiplication
"/" : (int, int) -> int            // integer division
```

The operator '+' *also* represents string concatenation (i.e., this one operator symbol is *overloaded*):

```
"+" : (string, string) -> string   // string concatenation
```

The operators '&' and '!' represent logical operations:

```
"&" : (bool, bool) -> bool         // logical and
"!" : (bool) -> bool               // logical not
```

The operators '=' and b '<' are overloaded to represent the comparisons between two values of the same type T (**int**, **string**, or **bool**):

```
"=" : (T, T) -> bool               // equality comparison
"<" : (T, T) -> bool               // less-than comparison
```

A **for** statement iterates over the consequent values from a specified integer range. The expressions specifying the beginning and end of the range are evaluated once only, at the beginning of the **for** statement. The **for** control variable behaves like a constant inside the loop: it cannot be assigned another value (before exiting the **for** statement). A control variable needs to be declared before its use in the **for** statement (in the global scope). Note that loop control variables are *not* declared inside **for** statements.

Context-free grammar for Mini-PL

The syntax definition is given in so-called *Extended Backus-Naur* form (EBNF). In the following Mini-PL grammar, the notation X* means 0, 1, or more repetitions of the item X. The '|' operator is used to define alternative constructs. Parentheses may be used to group together a sequence of related symbols. Brackets ("[" "]") may be used to enclose optional parts (i.e., zero or one occurrence). Reserved keywords are marked bold (as "**var**"). Operators, separators, and other single or multiple character tokens are enclosed within quotes (as: ". ."). Note that nested expressions are always fully parenthesized to specify the execution order of operations.

```
<prog>   ::=  <stmts>
<stmts>  ::=  <stmt> ";" ( <stmt> ";" )*
<stmt>   ::=  "var" <var_ident> ":" <type> [ ":=" <expr> ]
           |   <var_ident> ":=" <expr>
           |   "for" <var_ident> "in" <expr> ".." <expr> "do"
                  <stmts> "end" "for"
           |   "read" <var_ident>
           |   "print" <expr>
           |   "assert" "(" <expr> ")"

<expr>   ::=  <opnd> <op> <opnd>
           |   [ <unary_op> ] <opnd>

<opnd>   ::=  <int>
           |   <string>
           |   <var_ident>
           |   "(" expr ")"

<type>   ::=  "int" | "string" | "bool"
<var_ident> ::= <ident>

<reserved keyword> ::=
              "var" | "for" | "end" | "in" | "do" | "read" |
              "print" | "int" | "string" | "bool" | "assert"
```

Lexical elements

In the syntax definition the symbol <*ident*> stands for an identifier (name). An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

In the syntax definition the symbol <*int*> stands for an integer constant (literal). An integer constant is a sequence of decimal digits. The symbol <*string*> stands for a string literal. String literals follow the C-style convention: any special characters, such as the quote character (") or backslash (\), are represented using escape characters (e.g.: \").

A limited set of operators include (only!) the ones listed below.

'+' | '-' | '*' | '/' | '<' | '=' | '&' | '!'

In the syntax definition the symbol <*op*> stands for a binary operator symbol. There is one unary

operator symbol (<*unary_op*>): '!', meaning the logical *not* operation. The operator symbol '&' stands for the logical *and* operation. Note that in Mini-PL, '=' is the *equal* operator - not assignment.

The predefined type names (e.g.,"**int**") are reserved keywords, so they cannot be used as (arbitrary) identifiers. In a Mini-PL program, a comment may appear between any two tokens. There are two forms of comments: one starts with "`/*`", ends with "`*/`", can extend over multiple lines, and may be nested. The other comment alternative begins with "`//`" and goes only to the end of the line.

Sample programs

```
var X : int := 4 + (6 * 2);
print X;
```

```
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);
```

```
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;
```
