MiniPL Interpreter

Tuomas Tirronen

March 31, 2018

Contents

1 Solution architecture

The interpreter is divided in to components as described in figure 1.

Scanner reads the source code one character at a time, constructing valid tokens and passing them to the parser.

The parser matches the input token stream against the syntactic rules of the context-free grammar, and constructs an abstact syntax tree during the process. This solution does not build a parse tree, but instead it directly creates the abstract syntax tree.

The scanning and parsing process is performed in one pass: The parser requests new tokens from the scanner, which then processes the source code and returns the next token. When the parser is finished, the abstact syntax tree is retrurned to the main program.

Semantic analyzer analyzes the semantic correctness of the abstact syntax tree, e.g. allowed operations, type checking and performs the variable declaration (creates an entry in the symbol table). Semantic analyzer implements the visitor pattern to traverse the abstract syntax tree.

Interpreter also implements the visitor pattern and it interprets each node in the abstact syntax tree. The interpretation is relatively simple, as the evaluation and symbol lookups are done in the Evaluator component.

The evaluator is responsible for evaluating the different expressions in the source program. Evaluator also implements the visitor pattern.

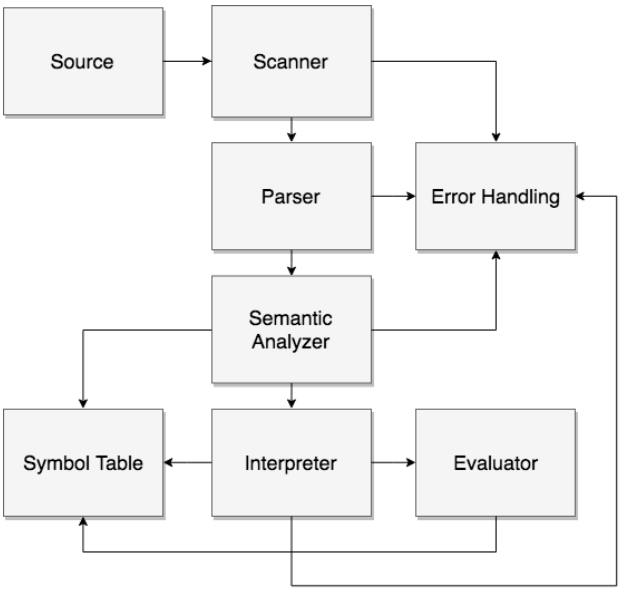

Figure 1: Overall architecture of the interpreter.

2 Token patterns

$integer \Rightarrow digit\ digit*$
$boolean \Rightarrow 0 \mid 1$
$string \Rightarrow letter\ letter*$
$identifier \Rightarrow letter\ (letter \mid digit)*$
$letter \Rightarrow A \mid B \mid ... \mid Z \mid a \mid\ b \mid ... \mid z$
$digit \Rightarrow 0 \mid 1 \mid 2 \mid ... \mid 9$

3 Context-free grammar

$program \Rightarrow block$
$block \Rightarrow statement\ (statement)*$
$statement$
$\Rightarrow$ "var" $identifier$ " : " $type$ [" := " $expression$]
$\Rightarrow$ $identifier$ " := " $expression$
$\Rightarrow$ "for" $identifier$ "in" $expression$ ".." "do" $block$ "end" "for"
$\Rightarrow$ "$read$" $identifier$
$\Rightarrow$ "$print$" $expression$
$\Rightarrow$ "$assert$" "("$expression$")"
$expression \Rightarrow term\ ((" + " \mid " - ")\ term)*$
$term \Rightarrow factor\ ((" * " \mid "/" \mid " < " \mid "\&" \mid " = ")\ factor)*$
$factor \Rightarrow integer \mid string \mid boolean \mid identifier \mid "(" expression ")" \mid "!" factor$

4 Abstract syntax trees

The abstract syntax tree is composed of different types of nodes, all inherited from an abstact node class.

Node types *BinOpNode*, *UnOpNode*, *IntNode*,*StrNode*, *BoolNode* and *IdNode* are constructed with the token information. The information of the *value* attribute varies between node types. For operation nodes (*BinOpNode* and *UnOpNode*), it stores the operation symbol (+, -, * or /). For *IdNode*, it stores the label of the symbol, and for literal nodes (*IntNode*, *StrNode*, *BoolNode*), it stores the value of the terminal as a string.

A node also stores information on its location in the source code. This is done by saving the row and column numbers that are obtained during the scanning process.

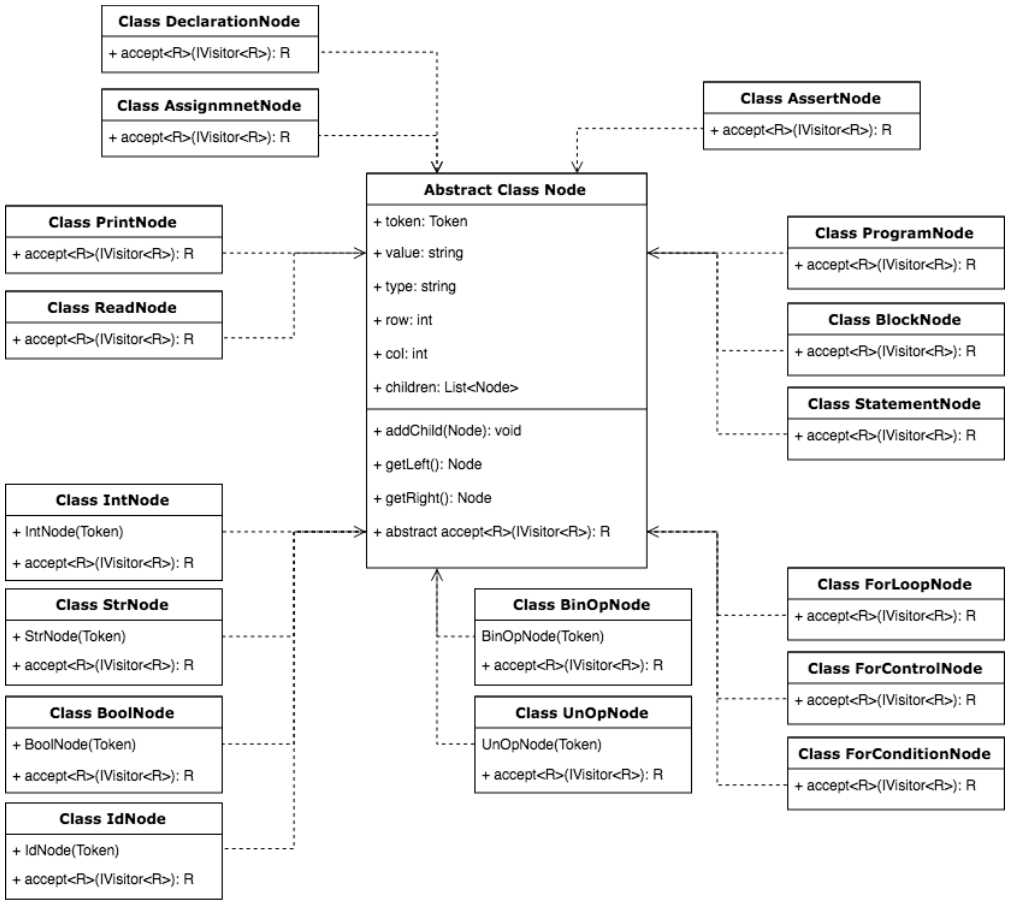

Figure 2: Class diagram of the abstact syntax tree.

The abstact syntax tree is constructed during the parsing process. As an example, for the following source program, an abstact syntax tree is build as shown in Figure 3.

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

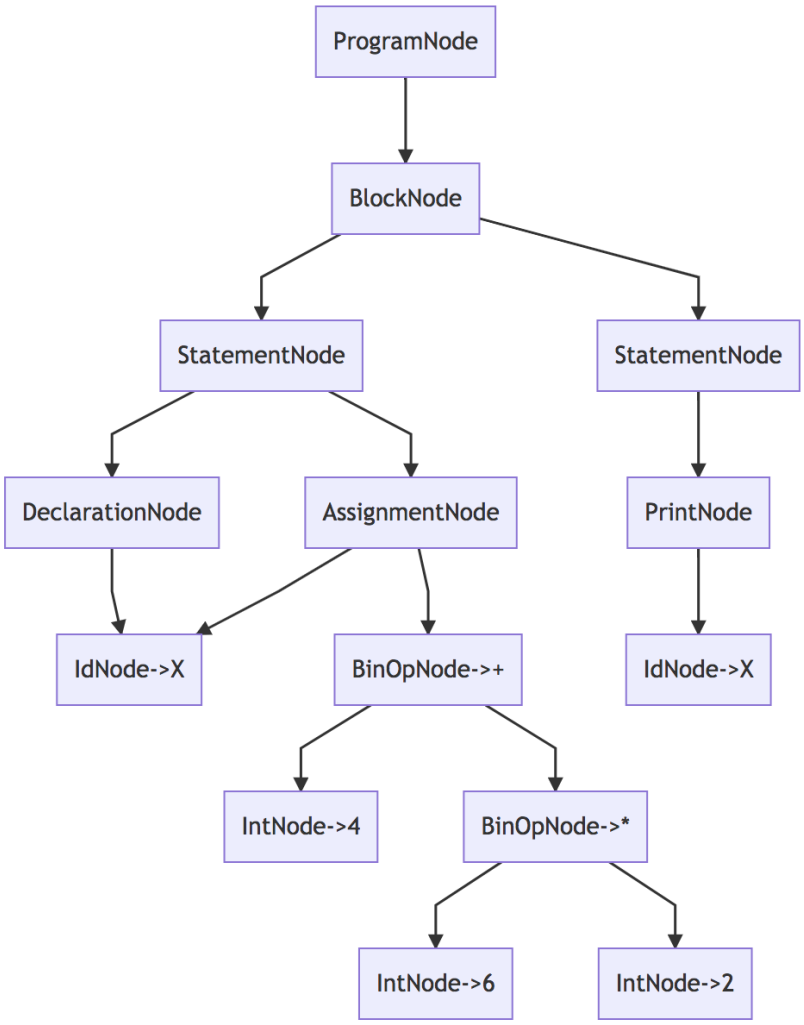

Figure 3: Example of an AST instance.

5 Error handling

Error handling component is used by scanner (lexical errors), parser (syntax errors), semantic analyzer (semantic errors) and interpreter (runtime errors). The aim for each component is to save the errors in a global error table, try to recover from it and continue their process; Scanner by returning a special unknown token and parser by starting again from the next statement (a token followed by *;*).

If the source program is found free from lexical and syntax errors, the program continue with the semantic analysis to ensure correct data types in operations and symbol usage. Once there are no semantic errors after the analysis, the source program will be interpreted. The interpreter will then detect runtime errors, that can be derived from user interaction, namely from providing invalid input for the *read* operation.

The evaluator assumes all the possible errors are found during previous components in the pipeline, so it does not make use of the error component.

6 Testing

7 Limitations

The known limitations and bugs for the interpreter are the following:

- Comment at the end of the source program will cause the scanner to break.
- The symbol table is not scoped
- Testing is not comprehensive
- Code is not properly commented

8 Usage

The interpreter requires to have Microsoft .NET environment installed. To interpret a MiniPL program, navigate to the *src* directory and run the following command:

`dotnet run` *your_program*

Compilers Project 2018: *Mini-PL interpreter* (24.01.2018)

Implement an *interpreter* for the Mini-PL programming language. The language analyzer must correctly recognize and process all valid (and invalid) Mini-PL programs. It should report syntactic errors, and then continue analyzing the rest of the source program. It must also construct an AST and make necessary passes over this program representation. The semantic analysis part binds names to their declarations, and checks semantic constraints, e.g., expressions types and their correct use. If the given program was found free from errors, the interpreter part will immediately execute it.

Implementation requirements and grading criteria

The assignment is done as individual work. When building your interpreter, you are expected to properly use and apply the compiler techniques taught and discussed in the lectures and exercises. The Mini-PL analyzer is to be written purely in a *general-purpose programming language*. C# is to be used as the implementation language, by default (if you have problems, please consult the teaching assistant). **Ready-made language-processing tools (language recognizer generators, regex libraries, translator frameworks, etc.) are not allowed**. Note that you can of course use the basic general data structures of the implementation language - such as strings and string builders, lists/arrays, and associative tables (dictionaries, maps). You must yourself make sure that your system can be run on the development tools available at the CS department.

The emphasis on one part of the grading is the quality of the implementation: especially its overall architecture, clarity, and modularity. Pay attention to programming style and commenting. Grading of the code will consider (undocumented) bugs, level of completion, and its overall success (solves the problem correctly). Try to separate the general (and potentially reusable) parts of the system (text handling and buffering, and other utilities) from the source-language dependent issues.

Documentation

Write a report on the assignment, as a document in PDF format. The title page of the document must show appropriate identifications: the name of the student, the name of the course, the name of the project, and the valid date and time of delivery.

Describe the overall architecture of your language processor with, e.g., UML diagrams. Explain your diagrams. Clearly describe yout testing, and the design of test data. Tell about possible shortcomings of your program (if well documented they might be partly forgiven). Give instructions how to build and run your interpreter. The report must include the following parts

1. The Mini-PL token patterns as *regular expressions* or, alternatively, as *regular definitions*.
2. A *modified context-free grammar* suitable for recursive-descent parsing (eliminating any LL (1) violations); modifications must not affect the language that is accepted.
3. Specify *abstract syntax trees* (AST), i.e., the internal representation for Mini-PL programs; you can use UML diagrams or alternatively give a syntax-based definition of the abstract syntax.
4. *Error handling* approach and solutions used in your Mini-PL implementation (in its scanner, parser, semantic analyzer, and interpreter).

For completeness, include the original project definition and the Mini-PL specification as appendices of your document; you can refer to them when explaining your solutions.

Syntax and semantics of Mini-PL (24.01.2018)

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 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.