Compilers Project 2020: Mini-pl interpreter documentation

Vili Lipo, 014814253

March 18, 2020

1 General view of the application

1.1 Architecture

The architecture of my implementation of Mini-pl interpreter follows closely the pipeline pattern presented on the lectures. The interpreter uses a multi-pass construction, as all of the parsing is done before semantical analysis, and all semantical analysis is done before the interpreting.

At the lowest level we have the class Source, that is responsible for reading a source file and giving characters from it one by one.

This is given as a constructor parameter to the class Scanner, that does the lexical analysis of the characters, and forms tokens out of them. The Scanner consists of a collection of routines that try to form a token by iterating the source. When a next token is asked from the scanner it screens out whitespace and comments and then it iterates through these routines until one of them returns a valid token. If no valid token is produced the scanner returns an error token.

When the source reaches the end-of-file the scanner produces end of file token.

The parser is a recursive descent parser that asks for the tokens one by one from the scanner. Every construct in the language has a own method for parsing it. These methods produce abstract syntax tree nodes.

The abstract syntax tree produced by the parser is first decorated by the TypeCheckVisitor, that checks for semantical errors and creates a symbol table based on variable declarations.

The table created by TypeCheckVisitor is then used by InterpretingVisitor that interprets the source file.

1.2 Testing

The classes Source, Scanner and Parser have quite comprehensive unit tests written to test their core functionality. Those can be found in the '/tests' folder. Also the 'tests/test.minipl' includes all the examples given in the language specification.

1.3 Shortcomings

1.4 Running the program

On Linux-machines running this program should be very straight forward. Open the directory where the archive was unzipped in a terminal. Then running

```
python main.py ./tests/test.minipl
```

2 Specifying the interpretation

2.1 Mini-PL token patterns

```
<integer> = <digit>*
<string_literal> = "<alnum>*"
<identifier> = ([a-z] | [A-Z])([a-z] | [A-Z] | _ | [0-9])*
<range> = \.\.
<keyword> = var | for | end | in | do | read
<keyword> =  print | int | string | bool | assert
```

2.2 Context-free grammar

```
<prog> = <stmnts>
<stmts> = <stmnt> ";" <stmnts>
<stmnts> = <epsilon> // {<epsilon>}
<stmnt> = "var" <var_ident> ":" <type> <assign_value>
<stmnt> = <var_ident> ":=" <expr>
<stmnt = "for" <var_ident> "in" <expr> ".." <expr> "do"
              <stmnts> "end" "for"
<stmnt> = "read" <var_ident>
<stmnt> =  "print" <expr> // {"print"}
<stmnt> "assert" "(" <expr> ")"
<assign_value> = ":=" <expr>| <epsilon>
<expr> = <opnd> <op> <opnd>
<opnd> = <int>
<opnd> = <string>
<opnd> = <var_ident>
<opnd> = "(" <expr> ")"
<type> = "int"
<type> = "string’’
<type> = "bool"
<var_ident> = <ident>
<reserved_keywords = "var" | "for" | "end" | "in" | "do" | "read" |
                     "print" | "int" | "string" | "bool" | "assert"
```

2.3 Abstract syntax tree

2.4 Error Handling

When the scanner encounters an error it sends an error token to the parser.

The parser uses context sensitive lookahead with exception driven error handling to recover from syntax errors.

Semantic errors are discovered by TypeCheckVisitor. These errors are printed to the user and prevent the interpreting process.

3 Work hour log