Compilers course project 2022 - Ossi Lehtonen

This is a project for course Compilers 2022 in University of Helsinki.
The program is written in C#.
The source code can be found from the folder 'src/MiniPLInterpreter'.

Date of delivery: 14.3.2022

Contents:

MiniPL Language

Please read MiniPL.pdf for detailed description of the language. This section is meant for answers required in the project setting (project_description.pdf)

Token patterns as regular expressions

OneCharTokens = \+|-|*|\/|< |=|&|!|\(|\)|;|:
Assignment = \:\=
String = \"[^\"]*\"
Int = \d*
Keyword = “var” | “for” | “end” | “in” | “do” | “read” | “print” | “int” | “string” | “bool” | “assert”
Ident = [A-Za-z]+(_|\d)*[A-Za-z]*

Token = Ident | Keyword | Int | String | Assignment | .. | OneCharToken

A modified context-free grammar suitable for recursive-descent parsing

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

<assignment> :=  “:=” <expr>

<expr>        ::= <opnd> [<expr_tail>]
                        | <unary_expression>

<expr_tail>   ::= <op> <opnd>
```

<unary_expression> ::= <unary_opnd> <opnd>

<unary_opnd> ::= “!”

<op> ::= '+' | '-' | '*' | '/' | '<' | '=' | '&'

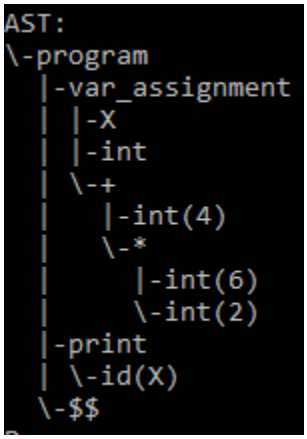
<opnd> ::= <int>
| <string>
| <var_ident>
| “(” <expr> “)”

<type> ::= “int” | “string” | “bool”
<var_ident> ::= [A-Za-z]+(_|\d)*[A-Za-z]*

ASTs

The programs are from MiniPL.pdf. The program also prints out AST of each program it interprets (if there is no error in the program).

Program 1 ast:



The syntax of the tree is as follows. “Program” is the root node. Node “var_assignment” is the first child of “program” and “print” the second. “X”, “Int” and “+” are the children of “var_assignment” and so on…

Program 2 ast:

See picture doc/asts/ast_2.JPEG

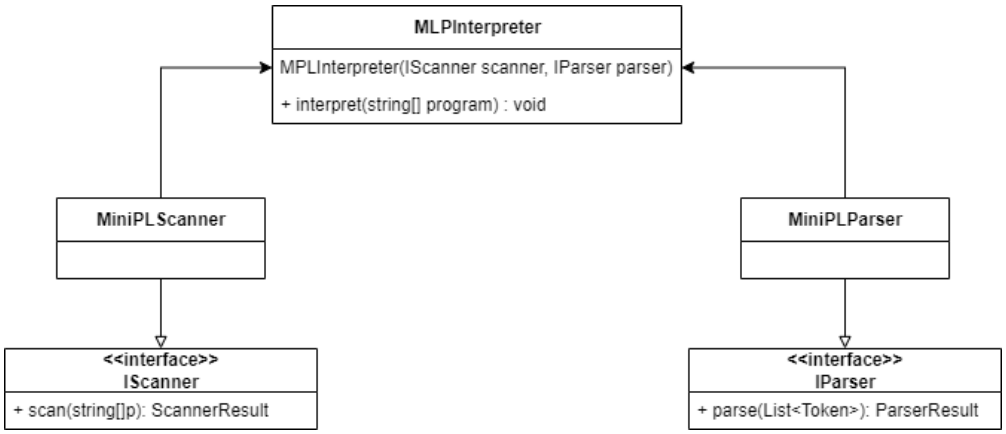
Program 3 ast:
See picture doc/asts/ast_3.JPEG

MiniPL Interpreter

In this section I try to briefly explain the structure of the program.
The actual source code is in src/MiniPLInterpreter.

High-level overview

The basic of the interpreter program is quite straightforward:



Note: This is a simplified picture of the structure of the program.

The MLPInterpreter class gets MiniPLScanner and MiniPLParser in its constructor. MiniPLScanner and MiniPLParser implement Interfaces which the MLPInterpreter class expects them to implement.

Interpreter class' "interpret"-function gets a list of strings as input. This list of strings is a MiniPL program, each string representing a row of a program.

The “interpret”-function calls Scanner’s “scan”-function, which returns the ScannerResult-object from the string list. The ScannerResult object contains a List of Tokens from the input program and a list of errors as strings.

Then, the Interpreter calls MiniPLParser’s “parse”-function with the List of Tokens as parameters. All of the non-helper functions in the MiniPLParser class correspond to the non-terminals of the CFG. The function returns a ParserResult-object. The ParserResult-object contains an AST and a list of errors.

Finally, if there are no errors from Scanner or Parser, the interpreter prints out the AST and then calls it’s private function “interpretAST”, which gets the AST as input, and the interpreting begins.

Parser functions correspond to the non-terminals of the CFG.

I will not go into more details of the implementation here. The basic implementation is explained above, but the Error Handling section opens up the implementation a bit more. I will also discuss the implementation in the shortcomings section.

Error handling

Scanner

The scanner does not do a lot of error handling. When an invalid character is seen, it adds an error to a list and skips the character. The error list is then returned to the MLPInterpreter, which logs the errors after parsing. The interpreting does not begin if there are errors noticed in the Scanner.

Parser

The parser error handling is a bit more comprehensive (and complicated) than the one in Scanner. The parser is responsible for handling both syntax and semantic errors.

There is a list of errors in the Parser, in which the errors are appended, and the error list is returned in ParserResult-object. If the MLPInterpreter notices errors in the list, it does not interpret the program but prints the errors.

The basic idea is, that if a parser subroutine sees an error (e.g it sees a token that it is not expecting, it throws a MiniPLException and adds it to the error list). After this, the parser goes to the next statement. However, there are some exceptions to this behavior, e.g:

1. If the var-function (responsible for handling variable assignment) recognizes that an invalid identifier is to be defined, it only adds an error to the error list and continues

normally. This is because the invalid identifier might be used later on the program and this way no cascading errors (e.g. undefined variable errors) are shown.
2. Errors in the for loop statement are handled so that the whole for loop is skipped.

Syntax errors

Each subroutine in the parser: var(), read(), print() and so on, are responsible for reporting syntax errors of the corresponding Non-Terminal. E.g. after the var() function has seen a valid identifier, it expects to see token ":" and calls a helper function:

```
miniPLHelper.checkTokenThrowsMiniPLError(CurrentToken(), ":");
```

The helper function throws a syntax error, and the error is catched by the calling function, which then handles the syntax error by skipping to the next valid statement.

Semantic errors

When identifiers are defined, the parser saves these identifiers to a C# dictionary:

```
public Dictionary<int, Dictionary<string, ParserIdentifier>> symbolTable;
```

The basic idea is to check from the dictionary whether the identifier exists when it is referenced. If it does not exist, a MiniPLException is thrown, which is then handled by going to the next statement.

The dictionary's int key corresponds to the current forLoopIndex. The values of the root Dictionary are Dictionaries with string keys and ParserIdentifier objects are the values. The ParserIdentifier -object contains information of the Identifier (type and whether it is a control variable). In the beginning of parsing, forLoopIndex is 0. When a forloop-statement is entered, forLoopIndex is incremented by one and a corresponding Dictionary is added to the symbolTable. And when a forloop-statement is exited, the forLoopIndex is decreased and the Dictionary is removed (making the identifiers defined in the loop as undefined).

The type checking is done in the following way. Operand(), expr() and unary_opnd() functions all return a type as a string. In expr() it is checked that all the operands in the expression are of the same type, and the operator is valid for the type, otherwise an error is thrown. In unary_opnd() it is expected that the following operand after "!" is of type boolean, otherwise an error is thrown.

Runtime errors

There are not a lot of runtime errors possible in the MiniPL language. However, one of them is related to the read-statement. The variable in which the read value is saved can be either string or int. If non-int is received when expecting a int, a runtime error is thrown and the MiniPL-program is terminated:

```
else if (node.value == "read")
{
    string identifierName = node.children[0].value;
    Operand identifier = Identifiers[identifierName];

    string readValue = consoleIO.ReadLine();

    if (identifier.type == "int")
    {
        if (!MiniPLHelper.isInt(readValue))
        {
            MiniPLExceptionThrower
                .throwMiniPLException($"Cannot cast {readValue} to type 'int'");
        }
    }
```

Testing

The src/MiniPLUnitTests folder contains automatic tests for the project.

The tests test the MLPInterpreter. There are 7 kinds of tests, each having a different MiniPL-program as input. It is tested that each of the programs are interpreted correctly (correct output is shown to the screen). In cases of errors, it is tested that correct error messages are shown.

The test cases contain the 3 example programs from MiniPL.pdf. In addition it contains the four following programs:

- A program with comments inside
- A program with errors (identifier starting with “_”, usage of undefined variable inside a loop, a row starting with an invalid statement)
- A program with a forloop with empty body (no statements inside the forloop)
- And A program with a nested forloop

Easiest way to run the automatic tests is by using Visual Studio.

The program can be obviously tested manually as well. See Running and Usage instructions.

Shortcomings

The project follows roughly the instructions gained from the course. The interpreter should work with all valid MiniPL-programs and incase of errors, they should be printed out. However, there are some shortcomings in the implementation:

- The Scanner outputs a List of tokens, rather than providing a function for Parser to fetch the next token. However, the scanner does not know anything about the parser, so the modularity is still there.
- The code is quite messy (at least in the parser). The task was surprisingly complicated and I should maybe have used more time on planning beforehand so that the code would be a bit more clear.
- There are some error messages printed that could be a bit more clear.
- The AST nodes are all strings. This made the processing and implementing the AST a lot easier, but

Running and Usage instructions

The program is written in C# and it has been tested with Windows 10 and Ubuntu 20.04 WSL, so it should work in the university environment.

The project uses .NET core 3.1, so it needs to be installed in the target machine. See install options from: https://docs.microsoft.com/en-us/dotnet/core/install/

I was able to install the 3.1 dotnet sdk to the linux environment with the commands:

sudo apt-get install -y gpg
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/ubuntu/{os-version}/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg

```
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-3.1
```

The source code can be found from folder 'src/MiniPLInterpreter'

Running

If Visual Studio is installed, one should be able to run the project easily with Visual Studio by opening the MiniPLInterpreter.sln solution in the src -folder.

If dotnet command line tools are installed, navigate to src/MiniPLInterpreter folder and run commands:

dotnet restore
dotnet run

Usage

When the program is running, it asks the user to provide the filename of a MiniPLProgram. It reads the MiniPLPrograms from folder src/MiniPLInterpreter/miniPL-programs. The files should be in .txt format. There are three example programs in the folder named 1.txt, 2.txt and 3.txt. These are the example programs from project_description.pdf.

Example:

```
Give the name of MiniPL program in folder (miniPL-programs):
1.txt
```

By pressing enter, the interpreting begins.

```
Give the name of MiniPL program in folder (miniPL-programs):
1.txt
AST:
\-program
  |-var_assignment
  | |-X
  | |-int
  | \-+
  |   |-int(4)
  |   \-*
  |     |-int(6)
  |     \-int(2)
  |-print
  | \-id(X)
  \-$$
Program:
------------
16
------------
```

The program first prints out the AST of the MiniPLProgram. The actual interpreted program is printed between two “--------”. In this case the MiniPL program prints out 16.

If there are errors on the MiniPL-program, they are printed out:

```
Give the name of MiniPL program in folder (miniPL-programs):
errors.txt
Errors:
Parser error: Invalid identifier '_nTimes' at row 1
Parser error: Undefined variable at row 7, col 8: Variable 'Y' is undefined
Parser error: Undefined variable at row 10, col 13: Variable 'sdjijasdjiasd' is undefined
Parser error: SYNTAX ERROR: UnExpected value '(' at row 10 expected ':='
```

Work Hours

Please see file Workhours.MD for more detailed info.
Total workhours: 82.