Documentation of the Mini-PL

A project for the Compilers course

Paavo Hemmo - 22.03.2020, 20:20

Running the interpreter

To start using Mini-PL, you should first download dotnet:
https://dotnet.microsoft.com/download.

Hopefully the installation will not cause too much problems. I myself use a Mac.

Move to the Interpreter directory in src/Interpreter/ and run the command:

dotnet run <absolute_path_to_file>

Where the <absolute_path_to_file> is the name of the file where you have inserted the Mini-PL code.

For example:

dotnet run /Users/user/code_file.txt

By default I left the code.txt file in src/Interpreter/ which can be used aswell, in case you don not want to create a new file.

-ast flag

The command also accepts a flag -ast, which you can position everywhere after dotnet run. For example:

dotnet run /Users/user/code_file.txt -ast
OR
dotnet run -ast /Users/user/code_file.txt

By adding the flag to the command, an Abstract Syntax Tree (AST) generated from the content of code_file.txt is printed.

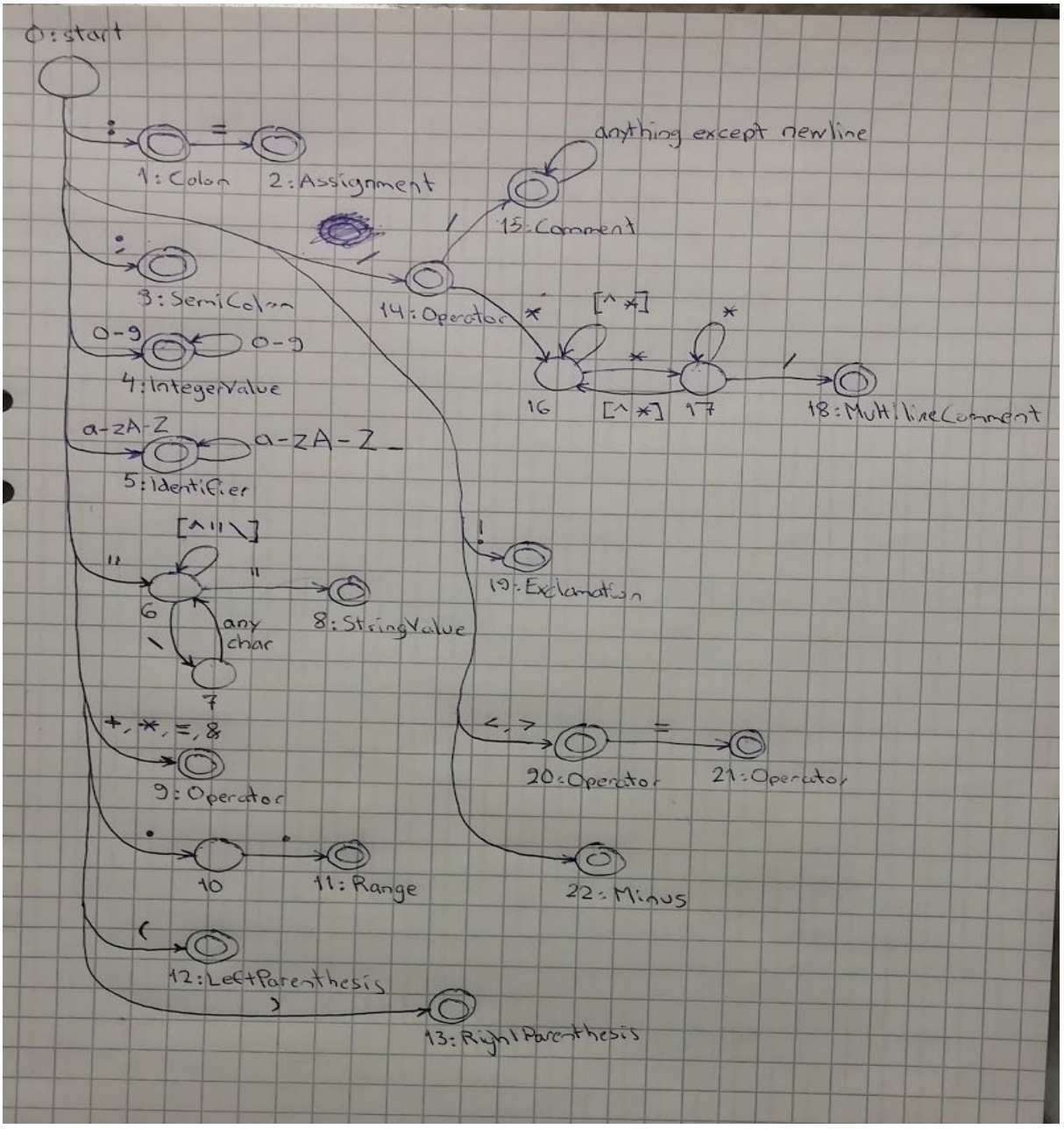
The AST is printed first, then the program is interpreted.

About the Mini-PL and project

I ran out of time writing this documentation, and could therefore not cover almost anything. COOL. Didn’t have time to comment the code either. I implemented the Typechecking at the same time the program is interpreted, because I wanted a language, where it is possible to get output from the program before an error occurs.

Mini-PL token patterns

The token recognition (Lexical analysis) is done with a Deterministic Finite Automata (DFA). The code of the automata is located in src/Interpreter/Lexer/DFAutomaton.cs. The code of the DFA is very simple, and it can clearly be seen that it implements the following DFA:



There might be something missing from the picture. Some notes about the automaton:

NewLine: `"\n|\r|\r\n"`
WhiteSpace: `"\\s"`
Are returned separately. For example NewLines are counted in the Scanner class.

Minus: "\\-|\\-"
For Minus signs there are two kinds of minuses recognized, which have different unicode values.

Known bugs in the Scanner

Because the unicode values might differ for some characters (like the minus), the Scanner will not recognize those characters and will therefore cause an error in lexical analysis.

Unimplemented parts

Escapes

The part for escaping characters in StringValues is implemented in the automaton, but not in the rest of the code. For example the following Mini-PL code works as such:

var x : string := “Trying \”to\” escape”;
print x;

The code above prints: Trying \”to\” escape
instead of: Trying ”to” escape .

Same with the the code:

var x : string := “newline\n”;
print x;

It only prints: newline without the wanted newline added at the end.

Nested MultilineComments

It is not possible to add nested multiline comments in my Mini-PL, which can be seen from the automaton.

The following code will cause an error:

/*/*nested*/*/

ERROR (line 1, column 12): Unexpected character: Operator

```
/*/*nested*/*/
           ^
```

The scanner will end the multiline comment after the first */ it discovers, and therefore it thinks the next character is the operator *. This error is thrown by the Parser.

Features

Use of whitespace

There can be whitespace anywhere in the code and that allows the programmer of the Mini-PL to write code as he/she likes. For example the following program is valid:

```
var
x:int:=3;print x

;
```

Multiline strings

In my implementation of the Mini-PL it is possible to write multiline strings. For example the following program is valid:

```
var x : string :=
“first line
second line
third line”;
print x;
```

And it prints:

```
first line
second line
third line
```

The correct line numbers and column numbers for tokens is handled the same way the MultilineComments are, and are therefore exact. For example adding an extra SemiColon at the end of the previous code will cause an error message with correct line and column numbers:

```
ERROR (line 5, column 8): Unexpected character: SemiColon

        print x;;
               ^
```

Context Free Grammar

The grammar of the Mini-PL can be expressed as the following EBNF:

<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> | <opnd>
<opnd> ::= “-” <int> | <string> | [“-” | “!”] <var_ident> | [“-” | “!”] “(“ <expr> “)”
<type> ::= “int” | “string” | “bool”
<reserved_keywords> ::= "var" | "for" | "end" | "in" | "do" | "read" | "print" | "int" | "string" | "bool" | "assert"

The grammar differs from the given grammar in the <expr> part. Instead of
<expr> ::= <opnd> <op> <opnd> | [<unary_op>] <opnd>

I added the “-” and “!” to be part of operands (<opnd>). This way it is possible to add minus or an exclamation point in front of single operands. The following codes are valid:

```
var y : bool;
var x : int := 2;
y := !((-x = -2) & !(1 = 2));
print y;
```

Prints false.

Abstract Syntax Tree

Below is the AST of a small program.

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

```
Program: (
  Statements: [
    (
      Declaration: (
        Identifier: nTimes,
        Type: IntegerValue,
        Token: (Value: nTimes, SymbolType: Identifier, Line: 1, Column: 4)
      )
    ),
    (
      Assignment: (
        Identifier: nTimes,
        Token: (Value: nTimes, SymbolType: Identifier, Line: 1, Column: 4),
        Expression: (
          UnaryExpression: (
            UnaryOperand: (
              Value: 0,
              Type: IntegerValue,
              Token: (Value: 0, SymbolType: IntegerValue, Line: 1, Column: 20),
              Negative: False,
              Not: False
            )
          )
        )
      )
    ),
    (
```

```
  Print: (
    UnaryExpression: (
      UnaryOperand: (
        Value: How many times?,
        Type: StringValue,
        Token: (Value: "How many times?", SymbolType: StringValue, Line: 2, Column: 6),
        Negative: False,
        Not: False
      )
    )
  )
),
(
  Read: (
    Identifier: nTimes,
    Value: ,
    Token: (Value: nTimes, SymbolType: Identifier, Line: 3, Column: 5)
  )
),
(
  Declaration: (
    Identifier: b,
    Type: IntegerValue,
    Token: (Value: b, SymbolType: Identifier, Line: 4, Column: 4)
  )
),
(
  ForLoop: (
    Assignment: (
      Assignment: (
        Identifier: b,
        Token: (Value: b, SymbolType: Identifier, Line: 5, Column: 4),
        Expression: (
          UnaryExpression: (
            UnaryOperand: (
              Value: 0,
              Type: IntegerValue,
              Token: (Value: 0, SymbolType: IntegerValue, Line: 5, Column: 9),
              Negative: False,
              Not: False
            )
          )
        )
      )
    ),
    Condition: (
```

```
  BinaryExpression: (
    Left: (
      UnaryOperand: (
        Value: b,
        Type: Identifier,
        Token: (Value: b, SymbolType: Identifier, Line: 5, Column: 4),
        Negative: False,
        Not: False
      )
    ),
    Operator: to,
    Right: (
      ExpressionOperand: (
        Negative: False,
        Not: False,
        BinaryExpression: (
          Left: (
            UnaryOperand: (
              Value: nTimes,
              Type: Identifier,
              Token: (Value: nTimes, SymbolType: Identifier, Line: 5, Column: 12),
              Negative: False,
              Not: False
            )
          ),
          Operator: -,
          Right: (
            UnaryOperand: (
              Value: 1,
              Type: IntegerValue,
              Token: (Value: 1, SymbolType: IntegerValue, Line: 5, Column: 19),
              Negative: False,
              Not: False
            )
          )
        )
      )
    )
  )
),
Statements: [
  (
    Print: (
      UnaryExpression: (
        UnaryOperand: (
          Value: b,
```

```
        Type: Identifier,
        Token: (Value: b, SymbolType: Identifier, Line: 6, Column: 8),
        Negative: False,
        Not: False
      )
    )
  )
),
(
  Print: (
    UnaryExpression: (
      UnaryOperand: (
        Value:  : Hello, World!\n,
        Type: StringValue,
        Token: (Value: " : Hello, World!\n", SymbolType: StringValue, Line: 7, Column: 8),
        Negative: False,
        Not: False
      )
    )
  )
),
(
  Declaration: (
    Identifier: z,
    Type: IntegerValue,
    Token: (Value: z, SymbolType: Identifier, Line: 8, Column: 6)
  )
),
(
  ForLoop: (
    Assignment: (
      Assignment: (
        Identifier: z,
        Token: (Value: z, SymbolType: Identifier, Line: 9, Column: 6),
        Expression: (
          UnaryExpression: (
            UnaryOperand: (
              Value: 0,
              Type: IntegerValue,
              Token: (Value: 0, SymbolType: IntegerValue, Line: 9, Column: 11),
              Negative: False,
              Not: False
            )
          )
        )
      )
```

```
    ),
    Condition: (
      BinaryExpression: (
        Left: (
          UnaryOperand: (
            Value: z,
            Type: Identifier,
            Token: (Value: z, SymbolType: Identifier, Line: 9, Column: 6),
            Negative: False,
            Not: False
          )
        ),
        Operator: to,
        Right: (
          ExpressionOperand: (
            Negative: False,
            Not: False,
            UnaryExpression: (
              UnaryOperand: (
                Value: 1,
                Type: IntegerValue,
                Token: (Value: 1, SymbolType: IntegerValue, Line: 9, Column: 14),
                Negative: False,
                Not: False
              )
            )
          )
        )
      )
    ),
    Statements: [
      (
        Print: (
          UnaryExpression: (
            UnaryOperand: (
              Value: Inside loop,
              Type: StringValue,
              Token: (Value: "Inside loop", SymbolType: StringValue, Line: 10, Column:
10),
              Negative: False,
              Not: False
            )
          )
        )
      ),
    ]
```

```
          )
        ),
      ]
    )
  ),
  (
    Assert: (
      BinaryExpression: (
        Left: (
          UnaryOperand: (
            Value: b,
            Type: Identifier,
            Token: (Value: b, SymbolType: Identifier, Line: 13, Column: 8),
            Negative: False,
            Not: False
          )
        ),
        Operator: =,
        Right: (
          UnaryOperand: (
            Value: nTimes,
            Type: Identifier,
            Token: (Value: nTimes, SymbolType: Identifier, Line: 13, Column: 12),
            Negative: False,
            Not: False
          )
        )
      )
    )
  ),
 ]
)
```

ForLoop

Looking at the AST's ForLoop's Condition nodes, it can be seen that the Operator in the BinaryExpression is "to". "to" is not a valid operator in the Mini-PL, and there is a reason why the operator is as such in ForLoops. When running the Interpreter, the "to" is replaced with either ">=" or "<=", based on the BinaryExpressions's Left and Right operands. This allows the ForLoop to be looped in either directions (either increasing or decreasing the for loop variable). For example the following program is valid:

```
var x : int;
for x in 0..-4 do
  print x;
```

end for;
And it prints:
0
-1
-2
-3
-4

Error Handling

The errors are implemented in src/Interpreter/Errors. At the moment there are only two kinds of errors: Error and DeclarationError. DeclarationError inherits the Error class, and Error class extends the Exception class of C#. This makes it possible to throw Errors. All the Errors are thrown and they are catched in the Program.cs file (The main method). Therefore whenever an Error occurs, the execution of the Program stops.

Testing

Testing is very bad at the moment. Only Scanner, Parser and InterpreterVisitor are tested, and they are not tested well.

Work hours

03.02.2020 | 2 h | C# kääntäjän asennus tietokoneelle (Mac). Scannerin toimintaa aloiteltu.
05.02.2020 | 2h | Scanneriin luotu toiminnallisuus tekstitiedoston jakamiseksi tokeneiksi. Tunnistaa regexien avulla nyt vain joitain tyyppejä, ei vielä esimerkiksi muuttujien nimiä.
08.02.2020 | 2h | Lisää avainsanoja määritelty kieleen. Tokenizer tallentaa nyt myös tokenin rivinumeron ja sijainnin rivillä.
09.02.2020 | 8h | Luotu ohjelmointikielelle CFG ja CFG:tä vastaava PDA. Tokenizer tunnistaa nyt
myös epävalidit merkit ja tulostaa rivit joilla on epävalideja merkkejä. Aloitettu koodamaan PDA:ta luokkaan Parser.
13.02.2020 | 4h | Aloiteltu Scannerin toteutusta DFA:na, päätin tehdä luentomateriaalia noudattaen, vaikka aikaisempi Scanner olikin toimiva.
14.02.2020 | 3h | DFA Scanneri toimii nyt. Virheraportteihin vielä oikeat rivinumerot.
15.02.2020 | 3h | Scannerin toimintaa parannettu.
16.02.2020 | 4h | Scanneri saatu kuntoon. Tallentaa nyt jokaiselle tokenille oikean rivinumeron ja sarakenumeron. Ottaa huomioon myös kommenteissa ja tekstiliteraaleissa olevat rivinvaihdot.
17.02.2020 | 1h | Perhedytty Syntaksiseen analyysiin. Toteutettu Parser hyvin yksinkertaiselle
EBNF kielelle.
17.02.2020 | 8h | Parser toteutettu alustavalle EBNF:lle. Scanneria jälleen paranneltu.

Luotu projekti dotnetilla ja lisätty tiedosto testaamiselle.
28.02.2020 | 2h | Virheiden käsittelyä toteutettu. Alustavasti pelkillä C#:n Exceptioneilla.
29.02.2020 | 6h | Virheiden käsittelyä paranneltu. Tulostuksessa nyt myös alkuperäisen tiedoston koodirivi. Hieman scriptiä toteutettu, jotta kääntäjän voi ajaa missä tahansa ja mille tiedostolle tahansa.
08.03.2020 | 3h | AST:n toteutusta aloitettu. Paljon pohdintaa, miltä AST näyttäisi.
09.03.2020 | 2h | AST.
10.03.2020 | 2h | AST.
11.03.2020 | 2h | AST rakentuu nyt Parserin tarkistaessa kielen syntaksia.
13.03.2020 | 5h | Symbolitaulu luotu ja aloiteltu koodin tulkkaamista.
14.03.2020 | 3h | Lähdekoodi kääntyy nyt oikein, mutta tulkin koodi on kamalaa.
16.03.2020 | 3h | Tyyppitarkastuksia ja koodin laatua paranneltu.
17.03.2020 | 3h | Tyyppitarkastuksia jatkettu edelleen.
18.03.2020 | 6h | Tyyppitarkastuksia ja parempia virheilmoituksia.
19.03.2020 | 10h | Tyyppitarkastuksia ja koodin kääntämistä.
20.03.2020 | 7h | Kieliopin hiomista ja tyyppitarkastuksia.
21.03.2020 | 8h | Kielen hiontaa. Testauksen toteutusta. Refaktorointia.
22.03.2020 | 10h | Refaktorointia. Dokumentaatiota.

Total: 121 hours