Mini-PL interpreter **22.3.2020**

CSM14204, Compilers

Hannu Örn
012095388

Instructions

Use the provided binaries in subdirectory *binary* to run the program in 64-bit Windows:

Mini-PL <arguments>

Pass the Mini-PL source file as an argument, for example:

Mini-PL ..\sample\times.mpl

Use the -AST command to print an abstract syntax tree:

Mini-PL -AST ..\sample\times.mpl

To build the program, open the solution file *Mini-PL.sln* in Visual Studio 2019. Or, use a command prompt to go to the directory of the solution file and run:

dotnet build

You can also build and run the program with one command:

dotnet run <arguments>

General architecture

This Mini-PL interpreter consists of four main components: scanner, parser, semantic analyser and interpreter.

The scanner initially reads the source file to memory. The source is then translated into a stream of tokens, one token at a time, as requested by the parser.

The parser is a recursive-descent predictive, top-down parser. An abstract syntax tree is created. Syntax errors in the source are handled with a context-sensitive error recovery technique.

The semantic analyser checks for data type validity, resolves data types of expressions and ensures correct use of variables. The AST is decorated with data type information and links from variable references to corresponding declarations.

After error-free parsing and semantic analysis, the interpreter executes the program.

Visitor design pattern is used to process the AST. TypeCheckVisitor implements the semantic analysis. ExecuteVisitor implements the interpretation, or execution of the program. PrintVisitor implements the AST printing feature.

Token patterns

Token types are defined in source file *Token.cs*, enum type **TokenKind**. The scanner recognises tokens according to the following patterns. Whitespace and (possibly nested) comments are skipped as per the Mini-PL specification.

```
Colon                ":"
Semicolon            ";"
OpenParenthesis      "("
CloseParenthesis     ")"
Plus                 "+"
Minus                "-"
Asterisk             "*"
Slash                "/"
Less                 "<"
Equal                "="
Ampersand            "&"
Exclamation          "!"
Assignment           ":="
RangeDots            ".."
int_Literal          [0-9]+
string_Literal       see below
bool_Literal         "false" | "true"
var_Keyword          "var"
for_Keyword          "for"
end_Keyword          "end"
in_Keyword           "in"
do_Keyword           "do"
read_Keyword         "read"
print_Keyword        "print"
int_Keyword          "int"
string_Keyword       "string"
bool_Keyword         "bool"
assert_Keyword       "assert"
Identifier           ([a-z] | [A-Z] | "_") ([a-z] | [A-Z] | "_" | [0-9])*
```

Double quotes are used for string literal delimiters. String literals must be closed before end of line. Backslash is an escape character. It can be used to represent three special characters: backslash itself as `\\` , a double quotation mark as `\"` or a newline as `\n` .

Modifications and amendments of the original Mini-PL

The following changes and amendments were made to the original Mini-PL specification:

- Abbreviated names for nonterminals were expanded (*operand* instead of *opnd*, *statement_list* instead of *stmts*). Nonterminal *op* was renamed to *binary_operator* to clarify the distinction between unary and binary operators.
- Literals of bool type, **false** and **true** were defined.
- Nonterminal *expression* was modified and a new nonterminal *expression_tail* was added to solve LL (1) issues.
- According to the original specification, ’each identifier may be declared once only’. This raises the question of declaring variables inside for loops. Statically speaking, such a variable is declared ’once only’, but from the viewpoint of running the code, the variable gets declared again on every cycle of the loop. A decision was made to allow such declarations. The interpreter deletes and creates the variable again on every cycle. The remaining value from the last cycle is carried outside the for loop.
- The input text handled by **read** statement is not whitespace-limited but newline-limited. A read statement always reads one line of text. In case of reading an integer, if the input text is not a valid integer, error message is printed and another line is immediately read.

Context-free grammar

```
<program>              ::=    <statement_list>

<statement_list>       ::=    <statement> ”;” ( <statement> ”;” )*

<statement>            ::=    <variable_declaration> |
                              <assignment> |
                              <for_statement> |
                              <read_statement> |
                              <print_statement> |
                              <assert_statement>

<variable_declaration>::=     ”var” <variable_identifier> ”:” <type> [ ”:=” <expression> ]

<assignment>           ::=    <variable_identifier> ”:=” <expression>

<for_statement>        ::=    ”for” <variable_identifier> ”in” <expression> ”..” <expression> ”do”
                              <statement_list> ”end” ”for”

<read_statement>       ::=    ”read” <variable_identifier>

<print_statement>      ::=    ”print” <expression>

<assert_statement>     ::=    ”assert” ”(” <expression> ”)”

<expression>           ::=    <operand> <expression_tail> |
                              <unary_operator> <operand>

<expression_tail>      ::=    <binary_operator> <operand> | ε

<operand>              ::=    <integer_literal> | <string_literal> | <bool_literal> |
                              <variable_identifier> | ”(” <expression> ”)”

<type>                 ::=    ”int” | ”string” | ”bool”

<variable_identifier> ::=     <identifier>

<reserved_keyword>     ::=    ”var” | ”for” | ”end” | ”in” | ”do” | ”read” |
                              ”print” | ”int” | ”string” | ”bool” | ”assert”

<binary_operator>      ::=    ”+” | ”-” | ”*” | ”/” | ”=” | ”<” | ”&”

<unary_operator>       ::=    ”!”

<integer_literal>      ::=    ([0-9])+

<string_literal>       ::=

<bool_literal>         ::=    ”false” | ”true”

<identifier>           ::=    ([a-z] | [A-Z] | ”_”) ([a-z] | [A-Z] | ”_” | [0-9])*
```

LL (1) conditions

The predict sets are trivially disjunct for all nonterminals except <expression>. The original grammar was modified and a nonterminal <expression_tail> was introduced to satisfy LL (1) conditions. Since <expression_tail> has an epsilon production, its First and Follow sets must be disjunct, which is clearly the case:

```
First(<expression_tail>) = { ”+”, ”-”, ”*”, ”/”, ”=”, ”<”, ”&” }
Follow(<expression_tail>) = { ”;”, ”..”, ”do”, ”)” }
```

Therefore, the grammar is LL (1).

Abstract syntax tree

The following is a list of classes that form the AST. Levels of indentation indicate class inheritance hierarchy. Abstract classes are in *italics*. Fields and their types are separated by a colon and listed immediately below the class name.

Classes ASTDummy_statement and ASTDummy_operand are used when erroneous source code prevents proper parsing of a statement or an operand.

Most of the fields are filled in by the parser. Two fields are filled in by the semantic analyser: *dataType* of AST_expression, indicating the data type of an expression, and *declaration* of AST_identifier, linking a variable reference to the corresponding variable declaration.

AST_node
- AST_program
  - statement_list : AST_statement_list
- AST_statement_list
  - statement_list: List<AST_statement>
- *AST_statement*
  - AST_assert_statement
    - expression : AST_expression
  - AST_assignment
    - identifier : AST_identifier
    - expression : AST_expression
  - AST_for_statement
    - identifier : AST_identifier
    - from : AST_expression
    - to : AST_expression
    - statement_list : AST_statement_list
  - AST_print_statement
    - expression : AST_expression
  - AST_read_statement
    - identifier : AST_identifier
  - AST_variable_declaration
    - identifier : AST_identifier
    - type : AST_type
    - expression : AST_expression
  - ASTDummy_statement
- *AST_expression*
  - *dataType : AST_type_kind*
  - *AST_operand*
    - AST_expression_operand
      - expression : AST_expression
    - AST_identifier
      - name : string
      - declaration : AST_variable_declaration

AST_integer_literal
value : string

AST_bool_literal
value : bool

AST_string_literal
value : string

ASTDummy_operand

AST_unary_operator
operand : AST_operand

AST_binary_operator
left : AST_operand
right : AST_operand

Example of an AST

The following AST can be printed with the -AST option.

```
program
   statement_list: 6
      variable_declaration
         identifier: nTimes : int_type
         type: int_type
         integer_literal: 0
      print_statement
         string_literal: "How many times?"
      read_statement
         identifier: nTimes : int_type
      variable_declaration
         identifier: x : int_type
         type: int_type
      for_statement
         identifier: x : int_type
         integer_literal: 0
         binary_operator: Minus, type: int_type
            identifier: nTimes : int_type
            integer_literal: 1
         statement_list: 2
            print_statement
               identifier: x : int_type
            print_statement
               string_literal: "   - Hello, World!"
      assert_statement
         binary_operator: Equal, type: bool_type
            identifier: x : int_type
            identifier: nTimes : int_type
```

Error handling

The scanner recognises three types of errors: invalid character, a single dot, and unterminated string literal. For an invalid character or a single dot, an error token is created to inform the parser about the error. For unterminated string literals, the scanner assumes a missing double quote at end of line and builds a normal string literal token. The scanner always saves row and column numbers in each token to be used for syntactic and semantic error messages later.

The parser uses context-sensitive error recovery technique. Follow sets are passed as an argument to parsing routines to allow minimal skipping of source code in case of parsing errors. A coherent AST

is always built, but some fields of AST nodes might be missing, and some expressions or operands might be dummies.

The semantic analyser resolves data types for all expressions and performs straightforward checks to find any type mismatches.

Due to the design, syntactic errors are reported before semantic errors. In other words, errors are not reported in the order they appear in the source code. An error report includes row and column numbers, a message describing the error, and the original line of source code combined with a pointer symbol indicating the exact location of the error. In order to prevent cascading error messages where the same error is reported at multiple levels of syntax hierarchy, only one message for each row/colum pair is printed.

If the scanner, parser and the semantic analyser did not find any errors, the interpreter is run. Only two types of errors can occur at runtime: assertion error and invalid integer input. In case of an assertion error, an error message accompanied with the original assert statement in the source code is printed. In case of an invalid integer, an error message is printed and a new integer is immediately read.

Sample programs

The project includes a collection of Mini-PL source files in the subdirectory *sample* (using file extension .mpl). Some of these are valid and working programs, others are demonstrations of the error reporting capabilities.

Testing

Manual testing was performed using the sample programs as input data. There are some ”debug prints” in the source code, especially the parser, that were used as a debugging aid. Visual Studio debugger was used extensively. There are no automated tests.

Shortcomings

The interpreter has only been somewhat superficially tested. It has no automated tests. There is room for improvement in architecture, modularity and general code cleanliness. Some refactoring would be appropriate if there was a need to develop the program further. However, the essential features seem to be working, and due to the extremely limited nature of Mini-PL language, I feel that the level of quality is adequate for this project.

Work hour log

date	hours	description
22.2.	6	set up dev tools, study C#
23.2.	5	implement scanner
24.2.	6	start implementing parser
25.2.	6	implement parser
12.3.	3	design AST
14.3.	6	implement AST and PrintVisitor
15.3.	6	start implementing ExecuteVisitor
16.3.	6	work on ExecuteVisitor, EvaluateVisitor
18.3.	6	semantic analysis
19.3.	6	parser error recovery, semantic errors
20.3.	6	improving error messages
21.3.	3	cleaning up, writing the report
22.3.	5	final tweaks
TOTAL	**70**	