COMPILERS - CSM14204

MINIPL DOCUMENTATION

March 22, 2020

Arttu Kilpinen
014013070
University of Helsinki
Department of Computer Sciences

INTRODUCTION

In the course Compilers - CSM14204 in university of Helsinki, the assignment was to implement an interpreter for the predefined programming language called MiniPL. MiniPL is a language designed for pedagogical purposes and the objective in the course was to learn the structure and implementation techniques of compilers and interpreters via the implementation of the MiniPL language. This document describes the architecture of the implementation, error handling and testing. The original assignment including the MiniPL grammar and its associated semantic rules can be found in the end of this document. Since the implementation were supposed to be done using an LL(1) parser, the original language definition are rewritten to be LL(1) compatible. Additionally of the new grammar specification, the lexical elements are defined using regular expressions. The last chapter includes the working log of the project.

DEFINITION OF THE LANGUAGE

Although the grammar defined in this chapter differs from the original one, the language resulting from the modified grammar is completely same as the one in the assignment. To implement an LL(1) parser of a language, the language needs to be an LL(1) compatible. The original grammar is mostly compatible but there exists two violations. Both of them are called a common prefix. The grammar rule

```
<expr> → <opnd> <op> <opnd>
       → <unary_op>  <opnd>
       → <opnd>
```

has a common prefix which means that two productions with the same left hand side begins with the same symbol. In this case the <opnd> terminal is in the *first* sets of both right hand sides and therefore in the *predict* set of both productions.

The second common prefix in the original grammar is the rule

```
<stmt> → "var" <var_ident> ":" <type> ":=" <expr> [ ":=" <expr> ]
```

which when represented in Backus-Naur form is equivalent to

```
<stmt> → "var" <var_ident> ":" <type> ":=" <expr>
       → "var" <var_ident> ":" <type>
```

The common prefix here is obvious. To eliminate the common prefixes the technique called left factoring can be used. The new grammar rules called *stmt_suffix* and *opnd_suffix* were introduced. The problematic rules described above can be represented as an LL(1) compatible grammar as follows:

```
<expr>        → <opnd> <opnd_suffix>
              → <unary_op>  <opnd>
<opnd_suffix> → <op> <opnd>
              → ϵ
<stmt>        → "var" <var_ident> ":" <type> <stmt_suffix>
<stmt_suffix> → ":=" <expr>
              → ϵ
```

The resulting grammar defines the same language as the original.

In this project not only those modifications were done but the grammar was simplified and restructured a little more. For example the new grammar does not include the <*var_ident*> defined in the original one. The <*expression*> rule is modified to use a new grammar rules <*binary_expression*> and <*unary_expression*>. Additionally the names of the rules are appended to full words (e.g. *opnd* in the original grammar is redefined to *operand*. The complete grammar used in this project is defined below.

Token patterns

The token patterns used in this project are described as regular expressions. The names of the tokens in this documents corresponds with the names in the source file *src/tokens.h*. The tokens described in this section are equivalent with the set of terminals in the implemented language.

Note that in the source code the binary operator is referred as a *TOKEN_BIN_OP*. However, in this document the binary operator is just called an operator.

This document uses the perl compatible regular expressions.

```
operator → + | - | * | / | = | < | &
identifier → [a-zA-Z][a-zA-Z0-9_]*
int_literal → [0-9]+
string_literal → \".*\"
type → int | string | bool
unary_operator → !
```

The string_literal has some additional rules. It can not contain a single backslash, nor an invalid control sequence. Since the assingment has some ambiguities (Which characters are in the set of "any special character"?) about the valid string literals, the implemented "special characters" were decided to be as follows: \\, \", \n, \t, \a, \b, \f, \r and \v.

In addition of this listing, there are many tokens with a single valid character sequence. Those are not listed here nor used in the following context-free grammar. (They are instead used as the double quoted value of the token. e.g. "for"). The tokens that are not listed here are all the keyword tokens which can only have a value of that keyword, parenthesis tokens, an assignment operator, a range token, a colon and a semicolon.

Note that in addition of the tokens described above, there are two special tokens: *eof* and *error*. The eof token is appended to the token list every time the lexer has finished the tokenization. Error token is added if the lexer can not construct any other token.

The modified context-free grammar

Here is the complete grammar used in the basis when the interpreter documented in this file was implemented. The parenthesized text describes the non-terminals and unparenthesized text describes the terminals. Some terminals are also denoted with a text inside as a double quotation marks. This comes handy when the terminal type in question only has one possible value (e.g. we write ";" instead of semicolon).

```
<program>            → <stmts> EOF
<stmts>              → <statement> ";" <stmts>
                     → ϵ
<statement>          → <declaration>
                     → <assignment>
                     → <for>
                     → <read>
                     → <print>
                     → <assert>
<declaration>        → "var" identifier ":" type <declaration_suffix>
<declaration_suffix> → ":=" <expression>
                     → ϵ
<assignment>         → identifier ":=" <expression>
<expression>         → <binary_expression>
                     → <unary_expression>
```

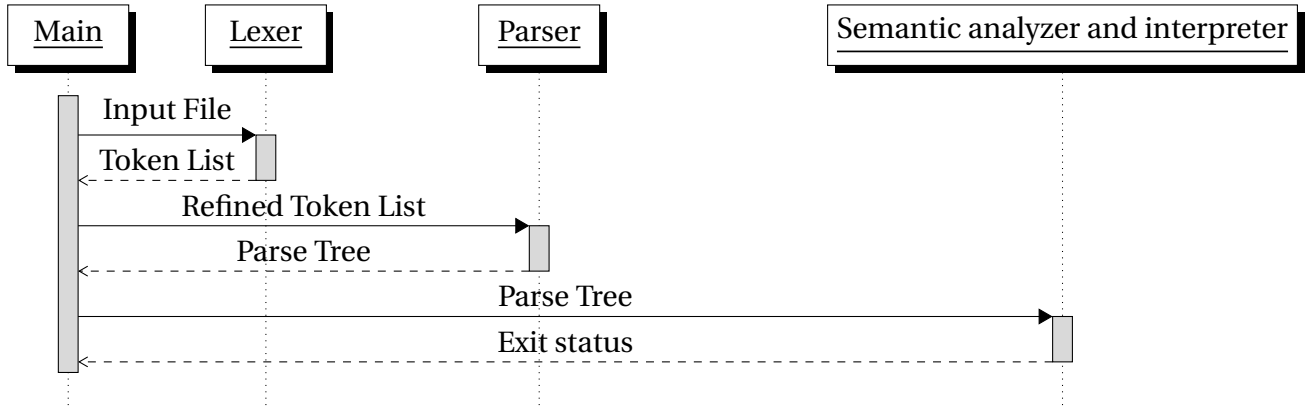
```
<binary_expression>    → <operand> <operand_suffix>
<unary_expression>     → unary_operator <operand>
<for>                  → "for" identifier "in"
                         <expression> ".." <expression>
                         "do" <stmts> "end" "for"
<read>                 → "read" identifier
<print>                → "print" <expression>
<assert>               → "assert" "(" <expression> ")"
<operand_suffix>       → operator <operand>
                       → ϵ
<operand>              → integer
                       → string
                       → identifier
                       → <enclosed_expression>
<enclosed_expression>  → "(" <expression> ")"
```

ARCHITECTURE

This MiniPL implementation is a one pass three phase broad interpreter. The phases are executed completely separately and the data flows from one phase to the next. The main program starts by opening the input file and invoking the lexer with the input file as an argument. The lexer creates the list of the tokens and returns the list to the main function. After that the main function discards the possible error tokens from the token list and prints error messages stored in the error tokens. The refined token list is then passed to the parser, which constructs the parse tree and returns it to the main function. If the parser encounters an error the NULL pointer is returned instead and the main function exits with an error status. Otherwise the parse tree is passed to the semantic analyzer and the interpreter is started. The following sequence diagram visualizes the rough structure of the interpreter.

There is also an object module *memory.o* which handles the memory allocations and deallocations. All three previously mentioned components use the memory service: The lexer needs the memory componen for (de)allocating memory for tokens. The parser needs to (de)allocace memory for the nodes in the parse tree and the semantic analyzer maintains a symbol table. Those requests for the memory component are not shown in the diagram, since the order of the function calls depends on the input data they are prosessing.

The lexer and the parser have a single responsibility of the phase they process. The



semantic analysis however is bundled with the interpretation. This decision was made to maintain the compactness of the code. Since the abstract syntax tree is not built and the parse tree is the internal data representation, there is no need to process it twice. The semantic analyzer and the interpreter must process the same parse tree so combining them seems reasonable.

Since the *program* is the start symbol of the MiniPL context-free grammar, the program node is the root of the parse tree. The parse tree consists of that node and possibly (if the input is not empty) sub trees associated with the program. The structure of the parse tree strictly corresponds to the grammar. The complete definition of the grammar can be found in this document and the equivalent source code is defined in the file *src/tree.h*

In addition of the parse tree the interpreter uses few data structures to represent its internal data.

The labels are used in the semantic analyzer / interprer phase. The label objects has a name and a value. The value consists of the value data and indicator variables that describes whether the value is error, empty and whether the value can be changed. Additionally the value contains a type identifier which defines how the data is used. The labels are stored in a singly linked list that is scanned every time a reference to a variable is made. The source code definitions of labels can be found in the file *src/label.h*

The token data type contains three values: the type (integer), the value (string) and the line number. Tokens are also stored in a singly linked list and the source code definitions of tokens is located in the file *src/tokens.h*

ERROR HANDLING

In any of the three phases the interpreter program may detect the input data to be not valid MiniPL program. In such an error situation the interpretation should not be continued or started even when the error could somehow be omitted. However, when encountering the error, the lexer and the parser should try to continue the process in order to be able to produce as many error messages as possible at a single run. This feature greatly improves the usability of the interpreter, since the user does not need to fix only one error at a time and try to recompile her code to get the next error. The lexer and the parser generated errors are cumulative. They are printed to the user and the compilation is continued if possible. The semantic analyzer / interpreter component however stops after the first error, since it makes no sense to continue the interpretation if the program were invalid.

The error handling in the lexer is is very straight forward. When the lexer detects the character sequence that does not match any other token, the error token is created and added to the token list. The error token includes all the characters from the errorneous point to the next whitespace character. For example if the input is otherwise correct but some of the identifiers is is started with an invalid character, then the whole identifier is considered to be error and not just the first charatcer. The value of the error token is also appended with a descriptive error message. The error tokens are inserted to the token list as any other tokens and the lexing is continued. After the token list is returned to the main function, the function *correctTokenlist* is called. This function reads the list from start to end and if errors are encountered the error messages are printed and the token is removed from the list. When the token list is passed to the parser, the error tokens are not in there anymore so the lexical errors does not yeld a parse errors.

The error handling in the parser is a bit more complicated. The parser must somehow recover from the error to be able to continue. The question is what grammar rule is the expected one when the previously parsed rule fails. The statements in the MiniPL are separated with semicolons and the *statement_list* grammar rule contains a *statement*, semicolon and another *statement_list*. This means that if the parse of some statement yelds an error, there is most probably a semicolon somewhere further. Indeed, this implementation discards all the tokens until the next semicolon, prints an appropriate error message and tries to continue the parsing if the error occurs in the statement subtree. Additionally the error message is printed with appropriate error message. If the error happens in the statement list after the previous statement is correctly parsed the semicolon is considered to be missing and the parsing of the next statement is starter. After the parse tree is generated the parser checks

the error indicator and decides whether interpreter phase can be starter. If there were any errors the parser deallocates the parse tree and returns NULL pointer to the main function, preventing the interpreter from starting. start.

The following is an example of the error output produced by the following code:

```
$ cat errors.mpl
read 3;
var % X : int := 4;
x := (3+3+3);
var s : string := "Hello World!\m";
print s
/* Unterminated comment..

$ ./minipl errors.mpl
Lexical error in line   2: Unidentified token: %.
Lexical error in line   4: Undefined control sequence \m in string literal.
Lexical error in line   6: Unterminated comment.
Syntax  error in line   1: Invalid read statement.
Syntax  error in line   3: Invalid assignment statement.
Syntax  error in line   5: Expected semicolon.
```

BUILD INSTRUCTIONS

There are several makefiles in the project directory structure. The makefiles are structured in the way that the lower level makefiles (the makefiles that are more close to the projects root directory) invokes the higher level ones. The actual build commands are in the higher level makefiles so the lower level makefiles only defines the order of the actual build commands. The actual interpreter has a makefile in the folder *src/* and running that makefile the interpreter is build and the executable binary file is moved to the folder *target/*. There are also a makefile for each test package (see the chapter Testing). The test makefiles compiles the binary used in one of the test packages.

The compilation is done by invoking the following command in the root directory of the project:

```
$ make
```

After the make is run the executable binary can be found in *target/minipl*. To run the MiniPL interpreter one must either move the binary to the local folder or invoke the interpreter from the target directory:

```
$ ./minipl <input_file>
```

or

```
$ path/to/minipl <input_file>
```

The makefile also contains the *clean* and the *clobber* rules. When make is invoked with a parameter *clean* the object code files (.o suffix) are deleted from the system. The *clobber* command first invokes the *clean* and then removes the binaries from the *target/* folders. (There is another target folder for the test binaries).

TESTING THE PROJECT

The implementation of the MiniPL interpreter was tested in three parts. As the implementation consists of three phases, this was found to be a natural approach. In addition of the *src/* directory the project root folder also contains the *tests/* directory. The tests for each part can be found there. The test directories has the following names: *tests/lex/*, *tests/parser/* and *tests/semantics/*. Every test package has their own main function that uses the external (interface) function of the particular phase. The main functions can be found in the directories *tests/src/lex/main.c*, *tests/src/parser/main.c* and *tests/src/semantics/main.c*.

The tests are not traditional unit tests that tests one function or a feature of a class separately. Instead the tests targets to one complete phase of the interpreter. The tests can be tought to be what is called an integration test. In the other words the lexer tests tests the integration of all functions used in the lexical analysis as a whole and the parser and semantic tests performs in the same manner.

Structure of the test packages

In each test package there are files called *test.cgf* and *test.sh*. The latter is a bash compatible shell script that runs the corresponding test binary *tests/target/<binary_name>* with a bunch of input files located in the *units/* folder of that test package. The *test.cfg* files are a simple files containing the input file name and the expected result, one per each line. The test script runs the input files with the test binaries and compares the actual results with those expected in the *test.cfg* file. Finally the test script prints the test file name and either the text "PASSED" or "FAILED" to the standard output.

Lexer tests

In the test package of the lexical analyzer the expected output of an input program is a sequence of the recognzed tokens. Every token has a corresponding numerical value in the source file *src/tokens.h*.

For example there is a test file *tests/lex/units/integration1.mpl* which contains the source code of the first sample program in the assignment:

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

The token types relevant for this test are in the following table:

Token type	Corresponding integer value
TOKEN_VARKEY	11
TOKEN_IDENTIFIER	6
TOKEN_COL	4
TOKEN_TYPEKEY	9
TOKEN_ASSIGN	5
TOKEN_INT_LIT	7
TOKEN_BIN_OP	0
TOKEN_LPAR	1
TOKEN_RPAR	2
TOKEN_SCOL	3
TOKEN_PRINTKEY	16

Therefore the test binary should produce the integer sequence "116495701707231663" when it is run with the example test file *integration1.mpl* as an input. Indeed, there is a correspongin row in the *test.cfg* for that:

```
$ grep integration1.mpl test.cfg
  integration1.mpl 116495701707231663
```

The test package of the lexer has a three types of tests. For every token type there is a test with an input file containin just that one token. This is a good basis to the other tests. In addition there are tests with a semi random token sequence with some token in the first, in the last and in the middle. The purpose of this is to test that tokens can be correctly recognized when they are in an arbitrary position and after or before some other tokens. Finally there are tests for the three sample programs from the assignment.

Note that the lexer ignores the comments so there is no such thing that comment token. To test comment recognition there are some comments (single line and multi line) added to the test files.

Note also that since the lexer only recognises tokens, the tested files can have arbitrarily random token sequences. The test files may or may not be valid MiniPL programs.

Parser tests

The parser is tested very similar way as the lexer. The test package contains the same files and a unit folder with the test unit files. The main function of the parser first invokes the lexer to get the list of the tokens. After that the lexer is run with that input. The test program then exits with either status 1 (success) or 0 (failed) depending on whether the input data were syntactically correct.

The test configuration file *test.cfg* contains the test file names and the expected result. The expected result is either 0 or 1. The test units contains one of every valid MiniPL statement. For example there are tests for a valid assert statement, for a valid print statement and for a valid for statement. Additionally there are test files with incorrect syntax. For example some tests lack of the semicolon between two statements and some tests are otherwise syntactically incorrect. Finally there are the same sample programs from the project assignment and they are tested to be valid.

Semantic tests

As the semantic analysis is bundled with the interpretation (see the chapter Architecture) the test package named “semantics” also tests for the runtime errors. The tests are performed the same way as the parser tests. There are *test.cfg* file containin the test unit names and the expected results, which can be either zero (failure) or one (success). The only difference is that there is also a third (optional) field that describes the user input. This feature is only used when the test file contains a read statement. For example the test named *error_read_int_incorrect_format.mpl* is otherwise valid program but it is tested with the input “incorrect_format”, which is a string so no integer value can be read. (the variable in the read statement is of type int). This test expects the runtime error and thus the expected value is zero.

Because the semantic analysis and the interpretation are executed last, the tests in this package most likely also catches the errors in the lexer and parser. For this reason this test package is the biggest one and contains more tests than the two previously described packages combined. There are tests for many successful programs and practically every possible runtime- or semantic error.

Running the tests

The whole testing process is fully automated. The makefile in the test root folder invokes the test scripts for every test package. It compiles the test binaries to the *tests/target/* folder (with names *lex_test*, *parser_test* and *semantics_test*) and launches the corresponding test.sh scripts in every test package. This proces can be invoked from the project root directory simply by running

```
$ make tests
```

WORK LOG

The work was done during and after the course (the course ended few weeks before the deadline of this project). The implementation was written with the same sequence as the processing of the data in the interpreter: The lexical analyzer was done first and the semantic analyzer last.

The lexical and the syntax analyzer was done quite early during the course. The actual interpreter and fully working solution was achieved about few weeks ago. However, after the

interpreter could fully recognize and run the input data, there was still many improvements to do. The commenting of the code and the layout and structure of the error message handling was done after everything else.

The complete work log is as follows:

Working day	Spent hours	What was done
15.2.2020	5h	The lexer was implemented
16.2.2020	3h	The tests for the lexer
21.2.2020	9h	The parser was implemented
3.3.2020	2h	The tests for the parser
6.3.2020	2h	Improved tests for parser
8.3.2020	4h	Semantic analyzer and interpreted started
10.3.2020	4h	Semantic analyzer partly done, sill some bugs
11.3.2020	4h	Semantic analyzer and tests for it
12.3.2020	2h	More tests for semantic analyzer
14.3.2020	4h	Refactoring and error message improvements
15.3.2020	3h	Making error messages partly cumulative in the lexer and the parser
20.3.2020	7h	Commenting the code and refactoring, the implementation is now done
21.3.2020	8h	Writing this document
22.3.2020	5h	Writing this document
Total	62h	The project is done

CONCLUSION

This implementation of the MiniPL was done mostly according to the assignment. Since the assignment have a few ambiguities the implementation decisions were freely made by those parts.

The first issue in the given MiniPL specification is the default values of the uninitialized variables, which are not specified. In this implementation the default values for integer, string and boolean types was chosen to be 0, "" (empty string) and false, respectively.

The second issue is how the less than operator (<) works with different data types. Obviously, the integer comparison follows the normal mathematical rules, but the operator is also speficied for string and boolean types. The implementation compares the string types so the string1 is less than string2 if it occurs first in alphabetical order. The boolean false value was chosen to be less than the boolean true value.

The specification of the print statement does not mention whether the implicit line break should be printed. The sample programs uses the implicit \n character only once, although it would be logical to use more often. In this implementation the implicit linebreaks are not printed.

The MiniPL specification states that any special character are represented using the escape character sequence. Since this set of special characters are not defined, this implementation only accepts \\, \", \n, \t, \a, \b, \f, \r and \v (See the section: Token patterns).

In this implementation the maximum token length is restricted to 1024 characters. This is in contradiction with the assignment's specification, which has no such limitation. This value is defined in the file *src/tokens.h* and it can be easily changed, if needed.

Compilers Project 2020: *Mini-PL interpreter* (Updated 03.03.2020)

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).
5. Include your *work hour log* in the documentation. For each day you are working on the project, the log should include: (1) date, (2) working time (in hours), (3) the description of the work done. And finally, the total hours spent on the project during the project course.

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.

Delivery of the work

The final delivery is due at 23 o'clock (11 p.m.) on Su **22.03.**2020. After the appointed deadline, the maximum points to be gained for a delivered work diminishes linearly, decreasing two (2) points per each hour late.

The work should be returned to the exercise assistant via e-mail, in a zip form. This zip (included in the e-mail message) should contain all relevant files, within a directory that is named according to your unique department user name. The deliverable zip file must contain (at least) the following subfolders.

```
<username>
  ./doc
  ./src
```

When naming your project (.zip) and document (.pdf) files, always include your CS user name and the packaging date. These constitute nice unique names that help to identify the files later. Names would be then something like:

project zip: `username_proj_2020_03_22.zip`
document: `username_doc_2020_03_22.pdf`

More detailed instructions and the requirements for the assignment are given in the exercise group. If you have questions about the folder structure and the ways of delivery, or in case you have questions about the whole project or its requirements, please contact the teaching assistant (Ilmo Salmenperä).

Syntax and semantics of Mini-PL (27.01.2020)

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 one 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.

Context-free grammar for Mini-PL

The syntax definition is given in so-called *Extended Backus-Naur* form (EBNF). In the following Mini-PL grammar, the notation X* means 0, 1, or more repetitions of the item X. The '|' operator is used to define alternative constructs. Parentheses may be used to group together a sequence of related symbols. Brackets ("[" "]") may be used to enclose optional parts (i.e., zero or one occurrence). Reserved keywords are marked bold (as "**var**"). Operators, separators, and other single or multiple character tokens are enclosed within quotes (as: ". ."). Note that nested expressions are always fully parenthesized to specify the execution order of operations.

```
<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>
           |  [ <unary_op> ] <opnd>

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

<type>   ::=  "int" | "string" | "bool"
<var_ident> ::= <ident>

<reserved keyword> ::=
              "var" | "for" | "end" | "in" | "do" | "read" |
              "print" | "int" | "string" | "bool" | "assert"
```

Lexical elements

In the syntax definition the symbol <*ident*> stands for an identifier (name). An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

In the syntax definition the symbol <*int*> stands for an integer constant (literal). An integer constant is a sequence of decimal digits. The symbol <*string*> stands for a string literal. String literals follow the C-style convention: any special characters, such as the quote character (") or backslash (\), are represented using escape characters (e.g.: \").

A limited set of operators include (only!) the ones listed below.

'+' | '-' | '*' | '/' | '<' | '=' | '&' | '!'

In the syntax definition the symbol <*op*> stands for a binary operator symbol. There is one unary

operator symbol (<*unary_op*>): '!', meaning the logical *not* operation. The operator symbol '&' stands for the logical *and* operation. Note that in Mini-PL, '=' is the *equal* operator - not assignment.

The predefined type names (e.g.,"**int**") are reserved keywords, so they cannot be used as (arbitrary) identifiers. In a Mini-PL program, a comment may appear between any two tokens. There are two forms of comments: one starts with "`/*`", ends with "`*/`", can extend over multiple lines, and may be nested. The other comment alternative begins with "`//`" and goes only to the end of the line.

Sample programs

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

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

```
print "Give a number";
var n : int;
read n;
var v : int := 1;
var i : int;
for i in 1..n do
    v := v * i;
end for;
print "The result is: ";
print v;
```
