# Compilers Project

Nicolas Hiillos 014801279

March 14, 2022

## Instructions

**cmake** and **gcc** are required to compile the interpreter. After `./build.sh` is run, to interpret a mini-pl progmram file, run `./build/mini-pl [filename]`. For testing purposes, the user can use `./build/mini-pl -s [filename]` to run merely the scanner, or `./build/mini-pl -p [filename]` to run the scanner+parser. Both commands print a readable result. Example programs are provided in `./test/`.

## Implementation details

### Scanner

The compiler uses a simple ad-hoc scanner and is based off the following regular expressions. The main functionality of the scanner consists of a long switch statement which applies these regular expressions one character at a time, looking ahead as necessary.

```
comment = "/*" ( non-* | "*" non-/ )* "*/"
        | "//" ( non-newline )* newline
string_lit = """ character character* """
slash = "/"
left_paren = "("
right_paren = ")"
minus = "-"
plus = "+"
asterisk = "*"
equal = "="
less = "<"
and = "&"
```

```
not = "!"
semicolon = ";"
assign = ":="
colon = ":"
range = ".."
var = "var"
for= "for"
end = "end"
in = "in"
do = "do"
read = "read"
print = "print"
int = "int"
string = "string"
bool = "bool"
assert = "assert"
boolean_lit = "false" | "true"
integer_lit = digit digit*
ident = letter ( digit | "_" | letter )*
```

The scanner handles errors by outputting tokens of type `ERROR` for unterminated strings and unexpected characters. These error tokens should then be handled by the parser.

## Parser

The parser implements each of the productions (everything enclosed in <>) in the CFG below as a class which is used to make up the syntax tree.

```
<prog> ::= <stmts>
<stmts> ::= <stmt> ‘‘;’’ ( <stmt> ‘‘;’’ )*
<stmt> ::= <var> | <assign> | <for> | <read> | <print> | <assert>
<var> ::= ‘‘var’’ <ident> ‘‘:’’ <type> [ ‘‘:=’’ <expr> ]
<assign> ::= <ident> ‘‘:=’’ <expr>
<for> ::= ‘‘for’’ <ident> ‘‘in’’ <expr> ‘‘..’’ <expr> ‘‘do’’ <stmts> ‘‘end’’ ‘‘for’’
<read> ::= ‘‘read’’ <ident>
<print> ::= ‘‘print’’ <expr>
<assert> ‘‘assert’’ ‘‘(’’ <expr> ‘‘)’’
<expr> ::= <opnd> <op> <expr>
            | [ <unary_opnd> ] <opnd>
<opnd> ::= <integer_lit> | <string_lit> | <boolean_lit> | <ident> | ‘‘(’’ <expr> ‘‘)’’
```

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

All infix operators are currently right-associative with equal precedence.

Each production of the CFG has a corresponding class, in `parser.cpp`. These classes can link to each other and form a syntax tree. This tree can then be traversed with the visitor pattern by classes inheriting the `TreeWalker` class. This parent class is used to implement a pretty printer of the syntax tree and the interpreter. The following excerpt from `parser.h` highlights the how the inheritance of these classes is derived from the CFG above. It also shows how the interface for the visitor class `TreeWalker` is implemented.

```
// parser.h
...
class TreeNode {
public:
  virtual void accept(TreeWalker *t) = 0;
};

class Opnd : public TreeNode {
public:
  void accept(TreeWalker *t) override { t->visitOpnd(this); };
};
class Int : public Opnd {
public:
  Scanner::Token value;
  Int(Scanner::Token v) { this->value = v; }
  void accept(TreeWalker *t) override { t->visitInt(this); };
};
class Bool : public Opnd {
public:
  Scanner::Token value;
  Bool(Scanner::Token v) { this->value = v; }
  void accept(TreeWalker *t) override { t->visitBool(this); };
};
class String : public Opnd {
public:
  Scanner::Token value;
  String(Scanner::Token v) { this->value = v; }
  void accept(TreeWalker *t) override { t->visitString(this); };
};
class Ident : public Opnd {
public:
  Scanner::Token ident;
  Ident(Scanner::Token v) { this->ident = v; }
  void accept(TreeWalker *t) override { t->visitIdent(this); };
};
class Expr : public Opnd {
public:
  Opnd *left;
  Scanner::Token op;
  Opnd *right;
  void accept(TreeWalker *t) override { t->visitExpr(this); };
};
class Binary : public Expr {
public:
  Binary(Parser::Opnd *left, Scanner::Token op, Parser::Opnd *right) {
    this->left = left;
    this->op = op;
    this->right = right;
  }
  void accept(TreeWalker *t) override { t->visitBinary(this); };
};
class Unary : public Expr {
public:
  Unary(Scanner::Token op, Parser::Opnd *right) {
    this->op = op;
    this->right = right;
  }
  void accept(TreeWalker *t) override { t->visitUnary(this); };
};
...
```

The code for these classes could be automatically generated, but I did not have enough time to do that.

The parser output is ready to be interpreted. No changes are made to the syntax tree. Below is a program and the AST made from it by the parser.

```
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);

// AST for the program above
(stmts
    (var ident:nTimes type:INT expr:(0))
    (print expr:("How many times? "))
    (read expr:nTimes)
    (var ident:x type:INT )
    (for ident:x from:(0) to:(nTimes - (1)) body:
        (print expr:(x))
        (print expr:(" : Hello, World!\n"))
    end for)
    (assert expr:(x = (nTimes)))
)
```

The statement following '(' in the AST corresponds to a production in the CFG and a class. `ident:[name]` and `type:[type]` are how identifier names and types are shown in a `var` statement. `expr:([expr])` is how expressions are printed. Expressions can be arbitrarily nested, as can for loops.

## Interpreter

The interpreter contains a class which inherits `TreeWalker`, used for traversing the AST. It uses a type named `Variable` to store and operate on mini-pl variables. It stores named variables in a map. Variables can be looked up by their name from the map. The interpreter also contains a stack used for unnamed variables and intermediate results in expressions.

Semantic analysis is handled by the interpreter dynamically. Variables are initialized as follows if no expression is provided; `bool:false`, `int:0`, and `string:""`. Variables can not be referenced, if they have not been initialized. Variable types are also checked by the interpreter.

## Future Improvements

- Generate scanner code

- Generate parser code
- Add operator precedence
- Finish REPL
- Handle parse errors gracefully
- Improve runtime error handling
- Perform semantic analysis ahead of time

## Work log

| Date | Hours | Progress |
|---|---|---|
| *[2022-02-08 Tue]* | 4 | Initialize repo and begin planning |
| *[2022-02-22 Tue]* | 2 | Setup cmake |
| *[2022-03-02 Wed]* | 5 | Plan and document scanner, begin implementation |
| *[2022-03-03 Thu]* | 8 | Implement scanner |
| *[2022-03-05 Sat]* | 7 | Begin implementation of parser |
| *[2022-03-08 Tue]* | 9 | Parser |
| *[2022-03-12 Sat]* | 2 | Finish parser, start interpreter |
| *[2022-03-13 Sun]* | 10 | Finish interpreter, docs |
| TOTAL HOURS | 47 | |

## Appendix A (MiniPL description)

## Syntax and semantics of Mini-PL (18.01.2022)

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_opnd> ] <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;
```

## Appendix B (Project description)

## *Compilers* Project 2020: *Mini-PL interpreter*

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 12:15 on Mo **14.03.**2022.

The work should be returned to the course Moodle page, 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 name. The deliverable zip file must contain (at least) the following subfolders.

```
<user>
  ./doc
  ./src
```

When naming your project (.zip) and document (.pdf) files, always include your 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: `user_proj_2022_03_14.zip`
document: `user_doc_2022_03_14.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 me.