CPS2000, CPS2006 - Compiler Theory and Practice
Course Assignment 2015/2016

Department of Computer Science. University of MALTA.
Sandro Spina

15th March 2016

Instructions

- This is **an individual** assignment. This assignment carries **50**% of the final **CPS2000** and **CPS2006** grade.
- While it is strongly recommended that you start working on the tasks as soon as the related material is covered in class, the firm submission deadline is **20**th **May 2016**.
- Hard copies **are not** required to be handed in. A soft-copy of the report and all related files (including code) must be uploaded to the VLE by the indicated deadline. All files must be archived into a single .zip file. It is the student's responsibility to ensure that the uploaded zip file and all contents are valid.
- You are to allocate approximately **32** to **40** hours for this assignment.

Description

This is the description for the assignment of units CPS2000 and CPS2006, Compilers: Theory and Practice. You are welcome to share ideas and suggestions. However, under no circumstances should code be shared among students. Please remember that plagiarism will not be tolerated; the final submission must be entirely your work. In this assignment you are to develop a lexer, parser and interpreter for the programming language *MiniLang*. The specification of this language can be found in the next sections. The assignment is composed of three major parts, a hand-crafted lexer and parser in C++, visitor classes to perform semantic analysis and execution, and a REPL (Read-Evaluate-Print-Loop) for the language. These tasks are explained in detail in the Task Breakdown section.

MiniLang is an expression-based strongly-typed programming language. The language has C-style comments, that is, `//...` for line comments and `/*...*/` for block comments. The language is case-sensitive and each function is expected to return a value. MiniLang has 4 types: 'real', 'int', 'bool', and 'string'. Binary operators, such as '+', require that the operands have matching types and the language does not perform any implicit/automatic typecast (coercing).

```
def funcSquare(x:real) : real {
        return x*x;
}

write funcSquare(2.5);
```

MiniLang (in EBNF)

⟨*Letter*⟩ ::= [A-Za-z]

⟨*Digit*⟩ ::= [0-9]

⟨*Printable*⟩ ::= [\x20-\x7E]

⟨*Type*⟩ ::= '`real`' | '`int`' | '`bool`' | '`string`'

⟨*BooleanLiteral*⟩ ::= '`true`' | '`false`'

1 ⟨*IntegerLiteral*⟩ ::= ⟨*Digit*⟩ { ⟨*Digit*⟩ }

2 ⟨*RealLiteral*⟩ ::= ⟨*Digit*⟩ { ⟨*Digit*⟩ } '.' ⟨*Digit*⟩ { ⟨*Digit*⟩ }

⟨*StringLiteral*⟩ ::= '"' { ⟨*Printable*⟩ } '"'

⟨*Literal*⟩
3 ::= ⟨*BooleanLiteral*⟩
4 | ⟨*IntegerLiteral*⟩
5 | ⟨*RealLiteral*⟩
6 | ⟨*StringLiteral*⟩

⟨*Identifier*⟩ ::= ('_' | ⟨*Letter*⟩) { '_' | ⟨*Letter*⟩ | ⟨*Digit*⟩ }

⟨*MultiplicativeOp*⟩ ::= '`*`' | '`/`' | '`and`'

⟨*AdditiveOp*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`or`’

⟨*RelationalOp*⟩ ::= ‘`<`’ | ‘`>`’ | ‘`==`’ | ‘`!=`’ | ‘`<=`’ | ‘`>=`’

6 ⟨*ActualParams*⟩ ::= ⟨*Expression*⟩ { ‘`,`’ ⟨*Expression*⟩ }

7 ⟨*FunctionCall*⟩ ::= ⟨*Identifier*⟩ ‘`(`’ [⟨*ActualParams*⟩] ‘`)`’

8 ⟨*SubExpression*⟩ ::= ‘`(`’ ⟨*Expression*⟩ ‘`)`’

9 ⟨*Unary*⟩ ::= (‘`+`’ | ‘`-`’ | ‘`not`’) ⟨*Expression*⟩

10 ⟨*Factor*⟩ 11 ::= ⟨*Literal*⟩
12 | ⟨*Identifier*⟩
13 | ⟨*FunctionCall*⟩
14 | ⟨*SubExpression*⟩
15 | ⟨*Unary*⟩

16 ⟨*Term*⟩ ::= ⟨*Factor*⟩ { ⟨*MultiplicativeOp*⟩ ⟨*Factor*⟩ }

17 ⟨*SimpleExpression*⟩ ::= ⟨*Term*⟩ { ⟨*AdditiveOp*⟩ ⟨*Term*⟩ }

18 ⟨*Expression*⟩ ::= ⟨*SimpleExpression*⟩ { ⟨*RelationalOp*⟩ ⟨*SimpleExpression*⟩ }

19 ⟨*Assignment*⟩ ::= ‘`set`’ ⟨*Identifier*⟩ ‘`=`’ ⟨*Expression*⟩

20 ⟨*VariableDecl*⟩ ::= ‘`var`’ ⟨*Identifier*⟩ ‘`:`’ ⟨*Type*⟩ ‘`=`’ ⟨*Expression*⟩

21 ⟨*FormalParam*⟩ ::= ⟨*Identifier*⟩ ‘`:`’ ⟨*Type*⟩

22 ⟨*FormalParams*⟩ ::= ⟨*FormalParam*⟩ { ‘`,`’ ⟨*FormalParam*⟩ }

23 ⟨*FunctionDecl*⟩ ::= ‘`def`’ ⟨*Identifier*⟩ ‘`(`’ [⟨*FormalParams*⟩] ‘`)`’ ‘`:`’ ⟨*Type*⟩ ⟨*Block*⟩

24 ⟨*WriteStatement*⟩ ::= ‘`write`’ ⟨*Expression*⟩

25 ⟨*ReturnStatement*⟩ ::= ‘`return`’ ⟨*Expression*⟩

26 ⟨*IfStatement*⟩ ::= ‘`if`’ ‘`(`’ ⟨*Expression*⟩ ‘`)`’ ⟨*Block*⟩ [‘`else`’ ⟨*Block*⟩]

27 ⟨*WhileStatement*⟩ ::= ‘`while`’ ‘`(`’ ⟨*Expression*⟩ ‘`)`’ ⟨*Block*⟩

⟨*Statement*⟩ 28 ::= ⟨*VariableDecl*⟩ ‘`;`’
29 | ⟨*Assignment*⟩ ‘`;`’
30 | ⟨*WriteStatement*⟩ ‘`;`’
31 | ⟨*ReturnStatement*⟩ ‘`;`’
32 | ⟨*FunctionDecl*⟩
33 | ⟨*IfStatement*⟩
34 | ⟨*WhileStatement*⟩
35 | ⟨*Block*⟩

36 ⟨*Block*⟩ ::= ‘`{`’ { ⟨*Statement*⟩ } ‘`}`’

37 ⟨*Program*⟩ ::= { ⟨*Statement*⟩ }

Task Breakdown

The assignment is broken down into five tasks. Below is a description of each task with the assigned mark.

Task 1 - Hand-crafted Lexer and Parser in C++

In this first task you are to develop the lexer and parser for the *MiniLang* language using the C++ programming language. The Lexer and Parser classes interact through the function *GetNextToken*() which the parser uses to get the next valid token from the lexer. The parser should be able to report any syntax errors in the input program. A successful parse of the input should produce an abstract syntax tree (AST) describing its' structure.

[Marks: 40%]

Task 2 - Generate XML of Abstract Syntax Trees (AST)

In OOP programming, the Visitor design pattern is used to describe an operation to be performed on the elements of an object structure without changing the classes on which it operates. In our case this object structure is the AST. For this task you are to implement a visitor class to output a properly indented XML representation of the generated AST.

E.g. var X : real = 8 + 2;

```
<Decleration>
    <Identifier>X</Identifier>
    <BinExprNode Op="+">
        <Real>8</Real>
        <Real>2</Real>
    </BinExprNode>
</Decleration>
```

[Marks: 10%]

Task 3 - Semantic Analysis Pass

For this task, you are to implement another visitor class to traverse the AST and perform type-checking. In addition to the global program scope, scopes are created whenever a block is entered and destroyed when control leaves the block. Note that blocks may be nested and that to carry out this task, it is essential to have a proper implementation of a symbol table.

[Marks: 5%]

Task 4 - Interpreter Execution Pass

For this task, you are to implement another visitor class to traverse the AST to execute the program. The *'write'* <Expression> statement in your test programs can be used to output the value of <Expression> to the console.

[Marks: 15%]

Task 5 - The REPL

The language is designed in such a way that one statement is a valid program by itself. This fact makes it easier for the interpreter (Task 4) to run in an interactive mode. Thus a REPL (Read-Execute-Print-Loop) environment can be implemented by creating a main class called, say MiniLangI (MiniLang Interactive), which acts as an interactive console class. MiniLangI is to wrap an instance of the parser and an instance of the symbol table.

When MiniLangI is started, a prompt is presented to the user for him/her to type in statements/expressions to be executed. Once the statement is input, MiniLangI is required to run the parser, the type-checker and the interpreter respectively and output the result of the computation. It is convenient that the interpreter maintains a special variable (in some languages this is usually called "it" or "ans" which holds the last result computed. Below is an example of an interactive session.

```
MLi> let x : real = 8 + 2;    // Creates variable 'x' and assigns 10 to it
Val ans : real = 10

MLi> 24 + 12;                 // Computes expression and stores it in ans
Val ans : int = 36

MLi> let y : bool = ans * 2;        // Error since the types do not match
Type mismatch:  real is assigned to a bool variable
```

Add a MiniLangI command to load scripts e.g.

```
MLi> #load "factorial.prog"
MLi> #st
```

Note that for any direct MiniLangI commands a parser is not required, a simple string comparison is enough. One can enhance MiniLangI with a number of commands/functions for example; #load (to load scripts), #quit (to end the session), #st (displays the contents of the symbol-table) This will aid in the debugging of your parser, type-checker and interpreter.

[Marks: 10%]

Report

In addition to the source and class files, you are to deliver a report. In your report include any deviations from the original EBNF, the salient points on how you developed the parser / interpreter (and reasons behind any decisions you took) including semantic rules and code execution, and any sample MiniLang programs you developed for testing the outcome of your compiler. In your report, state what you are testing for, insert the programs AST and the outcome of your test.

[Marks: 20%]

Final Notes

As an example, the MiniLang source script below, computes the answer of a real number raised to an integer power:

```
def funcPow( x : real, n : int ) : real
{
    var y : real = 1.0;        //Declare y and set it to 1.0
    if( n>0 )
    {
        while(n>0)
        {
            set y = y * x;        //Assignment y = y * x;
            set n = n - 1;        //Assignment n = n - 1;
        }
    }
    else
    {
        while(n<0)
        {
            set y = y / x;        //Assignment y = y / x;
            set n = n + 1;        //Assignment n = n + 1;
        }
    }
    return y;                          //return y as the result
}
```

Assuming that the above function is defined in a script file called power.prog, in MiniLangI we can make use of the function as follows:

```
MLi>#load "power.prog"

MLi>funcPow(3.0, 2);
Ans : real = 9.0

MLi>
```