CPS2000, CPS2006 - Compiler Theory and Practice
Course Assignment 2013/2014

Department of Computer Science. University of MALTA.

Gordon Mangion / Sandro Spina

20th January 2014

This is the description for the assignment of units CPS2000 and CPS2006, Compilers: Theory and Practice, which is worth 50% of the total mark for the respective units. The assignment has to be carried out on an individual basis. 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.

Deliverables

You will submit your project source code, executables and a PDF file containing the project documentation/report on optical medium. Note that assignment binaries must run straight from CD/DVD-ROM without the need for any installation unless specified in the accompanying document, in which case a detailed installation guide must also be provided.

Description

In this assignment you are to develop an interactive parser and interpreter for a simple scripting language called SXL (Simple eXpression Language). The specification of this language can be found in the next sections. The assignment is composed of three major parts, a hand-crafted parser, a parser generated by the JavaCC tool, and the latter is to be extended with a REPL (Read-Evaluate-Print-Loop) for the language. These tasks are explained in detail in the Task Breakdown section.

The SXL scripting language is an expression-based strongly-typed programming language. Each function returns the last evaluated expression. The language has C-style comments, that is, `//...` for line comments and `/*...*/` for block comments. The language is case-sensitive.

The SXL Grammar (in EBNF)

$\langle \mathit{Letter} \rangle$ ::= [A-Za-z]

$\langle \mathit{Digit} \rangle$ ::= [0-9]

```
⟨Printable⟩          ::= [\x20-\x7E]

⟨Type⟩               ::= ‘int’ | ‘real’ | ‘bool’ | ‘char’ | ‘string’ | ‘unit’

⟨BooleanLiteral⟩     ::= ‘true’ | ‘false’

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

⟨RealLiteral⟩        ::= ⟨Digit⟩ { ⟨Digit⟩ } ‘.’ ⟨Digit⟩ { ⟨Digit⟩ } [ ( ‘e’ | ‘E’ ) [ ( ‘+’ | ‘-’ ) ] ⟨Digit⟩
                         { ⟨Digit⟩ } ]

⟨CharLiteral⟩        ::= ‘'’ [ ⟨Printable⟩ ] ‘'’

⟨StringLiteral⟩      ::= ‘"’ { ⟨Printable⟩ } ‘"’

⟨UnitLiteral⟩        ::= ‘#’

⟨Literal⟩            ::= ⟨BooleanLiteral⟩
                      |  ⟨IntegerLiteral⟩
                      |  ⟨RealLiteral⟩
                      |  ⟨CharLiteral⟩
                      |  ⟨StringLiteral⟩
                      |  ⟨UnitLiteral⟩

⟨Identifier⟩         ::= ( ‘_’ | ⟨Letter⟩ ) { ‘_’ | ⟨Letter⟩ | ⟨Digit⟩ }

⟨MultiplicativeOp⟩ ::= ‘*’ | ‘/’ | ‘and’

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

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

⟨ActualParams⟩       ::= ⟨Expression⟩ { ‘,’ ⟨Expression⟩ }

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

⟨TypeCast⟩           ::= ‘(’ ⟨Type⟩ ‘)’ ⟨Expression⟩

⟨SubExpression⟩      ::= ‘(’ ⟨Expression⟩ ‘)’

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

⟨Factor⟩             ::= ⟨Literal⟩
                      |  ⟨Identifier⟩
                      |  ⟨FunctionCall⟩
                      |  ⟨TypeCast⟩
                      |  ⟨SubExpression⟩
                      |  ⟨Unary⟩

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

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

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

⟨Assignment⟩         ::= ‘set’ ⟨Identifier⟩ ‘<-’ ⟨Expression⟩
```

```
⟨VariableDecl⟩     ::= 'let' ⟨Identifier⟩ ':' ⟨Type⟩ '=' ⟨Expression⟩ ( ';' | [ 'in' ⟨Block⟩ ] )

⟨FormalParam⟩      ::= ⟨Identifier⟩ ':' ⟨Type⟩

⟨FormalParams⟩     ::= ⟨FormalParam⟩ { ',' ⟨FormalParam⟩ }

⟨FunctionDecl⟩     ::= 'function' ⟨Identifier⟩ '(' [ ⟨FormalParams⟩ ] ')' ':' ⟨Type⟩ ⟨Block⟩

⟨ReadStatement⟩    ::= 'read' ⟨Identifier⟩

⟨WriteStatement⟩   ::= 'write' ⟨Identifier⟩

⟨IfStatement⟩      ::= 'if' '(' ⟨Expression⟩ ')' ⟨Statement⟩ [ 'else' ⟨Statement⟩ ]

⟨WhileStatement⟩   ::= 'while' '(' ⟨Expression⟩ ')' ⟨Statement⟩

⟨HaltStatement⟩    ::= 'halt' [ ⟨IntegerLiteral⟩ | ⟨Identifier⟩ ]

⟨Statement⟩        ::= ⟨FunctionDecl⟩
                     | ⟨Assignment⟩ ';'
                     | ⟨Expression⟩ ';'
                     | ⟨VariableDecl⟩
                     | ⟨ReadStatement⟩ ';'
                     | ⟨WriteStatement⟩ ';'
                     | ⟨IfStatement⟩
                     | ⟨WhileStatement⟩
                     | ⟨HaltStatement⟩ ';'
                     | ⟨Block⟩

⟨Block⟩            ::= '{' { ⟨Statement⟩ } '}'

⟨Sxl⟩              ::= { ⟨Statement⟩ }
```

The SXL Type System

SXL has 6 types namely, 'int', 'real', 'bool', 'char', 'string' and 'unit'. The type 'unit', whose literal is written as the '#' (hash) sign, is used when an expression/statement does not have an actual value to return, equivalent to the 'void' type in C and Java. Command statements like 'write', for instance, return 'unit' type.

Binary operators, such as '+', require that the operands have matching types and the language does not perform any implicit/automatic typecast (coercing). Expressions have to use explicitly the typecast operator in order to resolve any type conflicts.

Task Breakdown

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

Task 1 - Hand-crafted Parser in C++

In this first task you are to develop the front-end parser for the SXL language. Use C++ to develop your lexical analyser and parser. The parser must generate an abstract syntax tree (parse-tree). Do not use any generators at this stage.

[Marks: 35%]

Task 2 - Build the parser using JavaCC

Create the Javacc grammar file for the SXL definition given above. You are free to modify the production rules as long as the changes are documented in the report and that the source language (SXL) remains unaltered. Should you prefer to use an alternative to the JJTree pre-processor in order to build the parse tree please go ahead.

[Marks: 15%]

Task 3 - Generate and Compare the Parse-Trees

You should enhance the parsers developed in Tasks 1 and 2 to output a textual (or XML) representation of the generated parse tree. Compare the parse-trees generated by the two parsers for your sample programs (use the same programs in Task 6).

E.g. Let X : integer = 8 + 2;

→ LetNode(Identifier(X), ExprNode(PlusNode(IntegerLiteral(8), IntegerLiteral(2))))

or

```
<LetNode>
        <Identifier>X</Identifier>
        <ExprNode>
                <PlusNode>
                        <IntegerLiteral>8</IntegerLiteral>
                        <IntegerLiteral>2</IntegerLiteral>
                </PlusNode>
        </ExprNode>
</LetNode>
```

[Marks: 10%]

Task 4 - Semantic Analysis and Execution

From this task onwards you are to enhance the JavaCC parser from **Task 2** only. In this task you are to use the visitor design pattern (or any method you deem suitable) to traverse the parse tree to perform type-checking and execute the parse tree nodes. Remember that it is essential to have a proper implementation of a symbol table which is used by both stages.

[Marks: 10%]

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 to run in an interactive mode. Thus a REPL (Read-Execute-Print-Loop) environment can be implemented by creating a main class called, say SXLI (Simple eXpression Language Interactive), which acts as an interactive console class. SXLI is to wrap an instance of the parser and an instance of the symbol table.

When SXLI 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, SXLI 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.

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

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

sxl> let y = ans * 2;        // Creates variable 'y' and assigns result
Val ans : int = 72

sxl> ans * 1.5;              // Error since the types do not match
Type mismatch:  integer and real!
```

You can add an SXLI command to load scripts e.g.

```
sxl> #load "factorial.sxl"
```

Note that for any direct SXLI commands a parser is not require, a simple string comparison is enough. One can enhance SXLI 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: 5%]

Task 6 - Sample Programs

Together with the above, you are to design and implement short sample source programs to test the outcome of your compiler. In your report, state what you are testing for, insert the programs parse tree and the outcome of your test.

[Marks: 15%]

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 SXL programs you developed for testing.

[Marks: 15%]

Final Notes

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

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

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

```
sxl>#load "power.sxl"

sxl>pow(3.0, 2);
Ans : Real = 9.0

sxl>
```