CPS2000 - Compiler Theory and Practice
Course Assignment 2019/2020

Department of Computer Science, University of Malta
Sandro Spina

March 10, 2020

Instructions

- This is **an individual** assignment. This assignment carries **50**% of the final **CPS2000** grade.
- It is strongly recommended that you start working on the tasks as soon as the related material is covered in class.
- The submission deadline is Sunday **24**th **May 2020**. A soft-copy of the report and all related files (including code) must be uploaded to the VLE by midnight of the indicated deadline. Hard copies **are not** required to be handed in. Source and executable files must be archived into a single .zip file before uploading to the VLE. It is the student's responsibility to ensure that the uploaded .zip file is valid. The PDF report must be submitted separately through the Turnitin submission system on the VLE.
- A report describing how you designed and implemented the different tasks of the assignment **is required**. Tasks (1-5) for which no such information is provided in the report will **not be** assessed.
- 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.
- You are to allocate approximately **40** hours for this assignment.

Description

In this assignment you are to develop a lexer, parser and interpreter for the small programming language - *SmallLang*. Your implementation can be carried out in either C++ or Java. The specification of *SmallLang* can be found in the next section. The assignment is composed of three major components: i) a FSA-based table-driven lexer and hand-crafted top-down LL(1) parser, ii) visitor classes to perform XML generation, semantic analysis and interpreter execution on the abstract syntax tree (AST) produced by the LL(1) parser and iii) a report detailing how you designed and implemented the different tasks. These three parts are further subdivided in tasks and explained in detail in the Task Breakdown section.

SmallLang 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. *SmallLang* has 3 primitive types: 'float', 'int' and 'bool'. Binary operators, such as '+', require that the operands have matching types and the language does not perform any implicit/automatic typecast. *SmallLang* supports the 'auto' type specifier both for variables and functions. For variables it specifies that the type of the variable that is being declared will be automatically deduced from its initialiser. For functions it specifies the return type will be deduced from its return statements. The 'auto' type specifier cannot be used to specify the type of the formal parameters in function declarations.

The following is a syntactically and semantically correct SmallLang program:

```
ff Square(x:float) : float {
 return x*x;
}

ff XGreaterThanY(x:float, y:float) : auto {
 var ans:bool = true;
 if (y > x) { ans = false; }
 return ans;
}

ff AverageOfThree(x:float, y:float, z:float) : float {
  var total:float = x + y + z;
  return total / 3;
}

var x:float = 2.4;
var y:auto = Square(2.5);
print y;                                  //6.25
print XGreaterThanY(x, 2.3);              //true
print XGreaterThanY(Square(1.5), y);      //false
print AverageOfThree(x, y, 1.2);          //3.28
```

SmallLang

The following rules describe the syntax of *SmallLang* in EBNF. Each rule has three parts: a left hand side (LHS), a right-hand side (RHS) and the '::=' symbol separating these two sides. The LHS names the EBNF rule whereas the RHS provides a description of this name. Note that the RHS uses four control forms namely sequence, choice, option and repetition. In a sequence order is important and items appear left-to-right. The stroke symbol (... | ...) is used to denote choice between alternatives. One item is chosen from this list; order is not important. Optional items are enclosed in square brackets ([...]) indicating that the item can either be included or discarded. Repeatable items are enclosed in curly brackets ({ ... }); the items within can be repeated **zero** or more times. For example, a *Block* consists of zero or more *Statement* enclosed in curly brackets.

⟨*Letter*⟩ ::= [A-Za-z]

⟨*Digit*⟩ ::= [0-9]

⟨*Type*⟩ ::= '`float`' | '`int`' | '`bool`'

⟨*Auto*⟩ ::= '`auto`'

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

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

⟨*FloatLiteral*⟩ ::= ⟨*Digit*⟩ { ⟨*Digit*⟩ } '`.`' ⟨*Digit*⟩ { ⟨*Digit*⟩ }

⟨*Literal*⟩ ::= ⟨*BooleanLiteral*⟩
| ⟨*IntegerLiteral*⟩
| ⟨*FloatLiteral*⟩

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

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

⟨*AdditiveOp*⟩ ::= '`+`' | '`-`' | '`or`'

⟨*RelationalOp*⟩ ::= '`<`' | '`>`' | '`==`' | '`<>`' | '`<=`' | '`>=`'

⟨*ActualParams*⟩ ::= ⟨*Expression*⟩ { '`,`' ⟨*Expression*⟩ }

⟨*FunctionCall*⟩ ::= ⟨*Identifier*⟩ '`(`' [⟨*ActualParams*⟩] '`)`'

⟨*SubExpression*⟩ ::= '`(`' ⟨*Expression*⟩ '`)`'

⟨*Unary*⟩ ::= ('`-`' | '`not`') ⟨*Expression*⟩

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

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

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

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

⟨*Assignment*⟩ ::= ⟨*Identifier*⟩ '`=`' ⟨*Expression*⟩

⟨*VariableDecl*⟩ ::= '`let`' ⟨*Identifier*⟩ '`:`' (⟨*Type*⟩ | ⟨*Auto*⟩) '`=`' ⟨*Expression*⟩

⟨*PrintStatement*⟩ ::= '`print`' ⟨*Expression*⟩

⟨*RtrnStatement*⟩ ::= '`return`' ⟨*Expression*⟩

⟨*IfStatement*⟩ ::= '`if`' '`(`' ⟨*Expression*⟩ '`)`' ⟨*Block*⟩ ['`else`' ⟨*Block*⟩]

⟨*ForStatement*⟩ ::= '`for`' '`(`' [⟨*VariableDecl*⟩] ';' ⟨*Expression*⟩ ';' [⟨*Assignment*⟩] '`)`' ⟨*Block*⟩

⟨*WhileStatement*⟩ ::= '`while`' '`(`' ⟨*Expression*⟩ '`)`' ⟨*Block*⟩

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

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

⟨*FunctionDecl*⟩ ::= '`ff`' ⟨*Identifier*⟩ '`(`' [⟨*FormalParams*⟩] '`)`' '`:`' (⟨*Type*⟩ | ⟨*Auto*⟩) ⟨*Block*⟩

⟨*Statement*⟩ ::= ⟨*VariableDecl*⟩ '`;`'
| ⟨*Assignment*⟩ '`;`'
| ⟨*PrintStatement*⟩ '`;`'
| ⟨*IfStatement*⟩
| ⟨*ForStatement*⟩
| ⟨*WhileStatement*⟩
| ⟨*RtrnStatement*⟩ '`;`'
| ⟨*FunctionDecl*⟩
| ⟨*Block*⟩

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

⟨*Program*⟩ ::= { ⟨*Statement*⟩ }

Task Breakdown

Task 1 - Table-driven lexer

In this first task you are to develop the lexer for the *SmallLang* language. The lexer is to be implemented using the table-driven approach which simulates the DFA transition function of the *SmallLang* micro-syntax. The lexer should be able to report any lexical errors in the input program.

[Marks: 20%]

Task 2 - Hand-crafted LL(1) parser

In this task you are to develop a hand-crafted predictive parser for the *MiniLang* 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 the structure of the program.

[Marks: 30%]

Task 3 - AST XML Generation Pass

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 (**not** the parse tree) produced by the parser in Task 2. For this task you are to implement a visitor class to output a properly indented XML representation of the generated AST.

```
let x : float = 3.142 * 2.0;

<Decl>
    <Var Type="float">x</Id>
    <BinExprNode Op="*">
        <FloatConst>3.142</FloatConst>
        <FloatConst>2.0</FloatConst>
    </BinExprNode>
</Decl>
```

[Marks: 5%]

Task 4 - Semantic Analysis Pass

For this task, you are to implement another visitor class to traverse the AST and perform type-checking (e.g. checking that variables are assigned to appropriately typed expressions, variables are not declared multiple times in the same scope, etc.). 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. Moreover, for this task your compiler will handle any type deductions resulting from the use of the 'auto' type specifier.

[Marks: 25%]

Task 5 - Interpreter Execution Pass

For this task, you are to implement another visitor class to traverse the AST and simulate an interpreter which executes the test program. The *'print'* <Expression> statement can be used in your test programs to output the value of <Expression> to the console and determine whether the computation carried out by the interpreter visitor is correct. Note that the symbol table can now be used to also store the values (in addition to type as was done with Task 4) of variables currently within scope.

[Marks: 20%]

Report

In addition to the source and class files, you are to write and submit a report. Remember that tasks 1 to 5 for which no information is provided in the report will not be assessed. In your report include any deviations from the original EBNF, the salient points on how you developed the lexer / parser / interpreter (and reasons behind any decisions you took) including semantic rules and code execution, and any sample *SmallLang* programs you developed for testing the outcome of your compiler. In your report, state what you are testing for, insert the program AST and the outcome of your test. As an example, the *SmallLang* source script below, computes the answer of a real number raised to an integer power:

```
//Function definition for Power
fn Pow(x:float, n:int) : auto
{
    var y : float = 1.0;          //Declare y and set it to 1.0
    if( n>0 )
    {
        for(; n>0 ; n=n-1)
        {
            y = y * x;            //Assignment y = y * x;
        }
    }
    else
    {
        for(; n<0 ; n=n+1)
        {
            y = y / x;            //Assignment y = y / x;
        }
    }
    return y;                     //return y as the result
}

let x : auto = Pow(2.1,10);

print x;  //prints to console 1667.988
```