Compilers and Interpreters
Take Home Test

Compilers and Interpreters

Honor Code

In accordance with The Harker School Honor Code, I certify that I have not received any unauthorized aid on this test. I further certify that I have not discussed this test with anyone, nor have I given any unauthorized aid. I promise not to discuss this test with anyone other than my teacher until it is handed back graded.

I understand that no collaboration of any kind is allowed. I will not discuss any aspect of this test with friends, parents, other students, former students or anyone other than my teacher. I will not share any work, directly or indirectly, using any electronic means (Google hangout or other social media). I understand that any violation of the honor code or collaboration policy will result in a 0 (zero) grade. I understand that my teacher may use software specifically designed to detect collaboration.

Name (printed) _____

Signature: _____

Date: _____

Score: _____/ 200

Consider the following grammar for a simple programming language we will call SIMPLE because it is, well simple. (Note that if you google SIMPLE, you will get a different language, one in which we have no interest.)

Program -> Statement P
P -> Program | e
Statement -> **display** Expression St1
| **assign id** = Expression
| **while** Expression **do** Program **end**
| **if** Expression **then** Program St2
St1 -> **read id** | e
St2 -> **end** | **else** Program **end**
Expression -> Expression **relop** AddExpr
| AddExpr
AddExpr -> AddExpr + MultExpr
| AddExpr – MultExpr
| MultExpr
MultExpr -> MultExpr * NegExpr
| MultExpr / NegExpr
| NegExpr
NegExpr -> -Value
| Value
Value -> **id** | **number** | (Expression)

The terminal symbols are identified by **bold** type.
A **relop** is defined by the regular expression **{<, >, >=, <=, <>,** =}. The symbol <> means 'not equal to'.
A **number** is defined as a non-negative integer.
The statement 'display 3 read x' will first display the number three and then prompt the user for input. The user input should be a number and it will then be stored as the new value x.

This choice of grammar may have several of consequences which you will need to explore. For example, your previous interpreter had a nonterminal called Condition that encapsulated a Boolean expression involving exactly two operands. The above CFG lumps these binary relationships into the Expression hierarchy (you may not re-introduce Condition). We are not necessarily interested in correcting any semantic difficulties caused by this CFG, but we must understand any of that are present thoroughly.

Recursive descent parsing with one token look-ahead is limited in that at each step of the parsing, there must be one and only one possible decision. Other parsing strategies are not so restrictive. A technique called left factoring is used as the first step in preparing a grammar for recursive descent parsing. The grammar for SIMPLE has been left factored for you. Remember that left factoring is only one step toward making a grammar suitable for recursive descent parsing. You must identify and perform all other steps, if any. When converting your grammar,

the new grammar must produce the same parse tree as the original grammar, and you must provide the analysis that demonstrates that this is true.

Your task is to complete the analysis, design, coding and debug of a SIMPLE interpreter. You should strive to reuse code from your PASCAL interpreter where appropriate. Your interpreter must produce an Abstract Syntax Tree as an intermediate representation and it must use a recursive descent parser. You may not use FLEX, JFLEX, LEX, YACC, CUP, BISON or any other automatic compiler/scanner/parser generator. You must follow the style guide.

Deliverables

You are to generate a formal report detailing your design, implementation, testing strategy, testing and results. It is to be written in the style of a research paper. English grammar counts.

It will be necessary for you to design and implement a large amount of code for this project. While I want to see the code, it is not the focus of your work. Attach your code as an appendix to your report.

You must turn in:

1. A complete design report for an interpreter that interprets the SIMPLE language organized as follows:
    a. Introduction describing the purpose of the document.
    b. Main body describing the overall design of your interpreter.
        1. Analysis of limitations and semantic issues present in the grammar.
        2. Grammar transformations, if any, must be shown along with the reasoning behind them
        3. Demonstration that the grammar transformations do not change the original grammar must be included.
        4. A design showing the decomposition into objects and the services each object will provide must be included.
        5. A complete test strategy must be included.
        6. There should not be any code in this section
    c. Test plan
    d. Test code printout
    e. Results
    f. Conclusions
    g. Source Code (appendix)

2. Source code submitted in a single zip archive to Athena2.
3. You will be asked to demonstrate your interpreter!

Due Date: On November 20, 2014 you will need to show me that you can run my test program. Your documents and code are due at the end of class on November 20 2014.