CS 352 Spring 2016

Compiler Projec 2

Posted Feb. 22, 2016, Due March 7, 2016, 11:59pm

This project is to be done by each student individually. Read the Academic Honesty Policy posted on Piazza carefully.

1. Objective and Scope

The main objective of this project is to get practice in (i) designing semantic actions; (ii) proper design of grammar rules to ensure correct order of operations; (iii) implementing an elementary symbol table; and (iv) implementing basic type checking and run time error detection. Project 2 also adds new production rules to the Yacc program developed in Project 1 such that new kinds of variables are supported in our *miniscript* language.

To interpret a script, in general the interpreter needs to build the abstract syntax tree (AST) and then traverse the tree to perform the encountered operations (sometimes repeatedly). However, in Project 2, we do not implement conditional statements and function calls, therefore all operations can be performed as the parser performs semantic actions during parsing. The script execution is done by issuing appropriate C/C++ statements in the semantic actions, as illustrated in lectures. Type checking may need to be performed immediately before certain operations are performed, as we will elaborate in later sections.

During the execution, all conventional associativity and precedence rules such as in C must be followed. Incorrect implementation will likely cause the output of the program to be incorrect and therefore fail the test.

Notice that the only thing visible as the output of running a perfectly correct miniscript program will be what is printed by document.write() statements. You will interpret such statements by calling printf() standard functions appropriately.

1. Language Features

We still have three kinds of statements as in Part 1: i) declaration statements; ii) assignment statements; and iii) document.write(….). In addition to the language features presented in Project 1, a few new features are introduced in Project 2. Implementation of some of the old languages features is also explained here.

1.1 Objects

Variable declaration in Project 1 declares scalar variables only. In Project 2, we add object declaration. Unlike JavaScript, we require all fields of an object to be declared before they can be accessed in assignment or document.write statements. An object declaration takes the following form:

var *ID* **= {**
<a list of fields>
}

In our test cases, "var ID = {" will appear exactly in a single line by itself. However, you are allowed to let "var ID = {" to follow other statements in the same line, but a line break must follow "{". The closing "}" must appear as exactly a single line by itself. The test cases will not have ";" following "}" but we allow a semicolon to follow "}" in the same line. These treatments are to make the handling of newlines simpler. At least one field must be in <a list of fields>, and <a list of fields> may occupy one or more lines. Two neighboring fields are separated by a comma (i.e. ","), which may or may not be followed by newlines. The last field must not be followed by a comma. A field may or may not have its value initialized. To simplify the implementation, a field will never be an object. Hence, the initialization is the following form:

ID : <expression>

In the above, <expression> is an expression that either has an integer value or is a string. (See below for operations on strings.) A field must not be separated into multiple lines. The following shows an example of valid object declarations:

```
var person = {
lastName:"Lincoln", firstName,

age: 150, eyeColor: "blue"+"grey"
}
```

A field's value can be initialized or modified by an assignment, such as "person.age = 100;"
Once a field is assigned a value, it can be used as an operand in a later statement just like a scalar variable, e.g. "x = person.age;" **In the rest of the specification, when we refer to a variable, it may be a scalar or a field name.**

Your code will need to look up the symbol table to find the object first and the field next.

In Miniscript, unlike in C, there are no type declarations for a scalar variable. Instead, a scalar has its type defined by the first value assigned to it after its declaration. In other words, the type definition is implicit and dynamic. To change a scalar variable's type, the variable must be re-declared by a "var" statement before a value of a different type may be assigned to it. Without such a re-declaration, a type error is generated. Similarly, the type of a field in an object is defined by the first value assigned to it. Once

assigned the type this way, any reassignment of the field's value must agree with the type, unless the entire object is re-declared by a "var " statement.

1.2 Scope of a Declaration

A pair of "{" and "}" is used to start a new scope of variable declarations as in the C language. Scopes may be nested. Variables declared in different scopes are different entities even if they have the same names. When a variable is referenced, the innermost scope enclosing the reference applies.

Within the same scope, a variable may be re-declared, with its type becoming undefined until its value is re-initialized.

Similar to object declarations, we specify the following form of scopes:

in our test cases, a new scope (also known as a compound statement) will always start a new line with "{", which is followed by another newline and then followed by a nonempty list of statements. However, to simplify the production rules, we do allow the new scope symbol "{" to follow other statements: e.g. "a=1; {" in the same line. Also, in our test cases, the closing "}" for any scope will occupy a line by itself, not to have other statements either before or after in the same line. However, we do allow statements to appear before and after "}" in your grammar. Finally, we allow a single ";" to follow "}" although the test cases will not have it.

1.3 Type Rules

The type rules in MiniScript will be somewhat stricter than JavaScript. (The reason the JavaScript has so loose type rules is the intent to avoid aborting the execution of a mobile code, but such relaxation also has caused criticisms because of the high possibility of hidden unintended errors.) A stricter rule also simplifies the execution. In this part of the project, your interpreter will report type rule violations, as if we are in the testing phase of the interpreter before deploying it in a browser.

We still have four operators, namely +, -, *, /, to form an expression. An operand can be a constant, a variable, or a sub-expression. A constant may be an integer or a string, and your interpreter must determine the correct type of the constant. A variable may be a scalar (which is just an ID) or an object's field. As stated above, a variable obtains its type from the first assignment of its value, which is an integer or a string. Hence your interpreter must record each variable's type for type checking. The operands of a binary operation must have the same type. Otherwise a type rule violation occurs. Using an object name as an operand causes a type violation. (This implies that we do not allow copying objects without copying individual fields.)

A variable must be declared before it can be referenced in the program. Modifying (write the value) or accessing (read the value) a variable that has not been declared by "var …" statement is a violation of the type rule.

Between integer values, +, -, *, / always produce integer results. Between two strings, the + operation concatenate the two strings together, e.g. “ab c” + “cdd ee f” becomes “ab ccdd ee f”. Similarly, person.firstName + “ “ + person.lastName will have the value “Abe Lincoln”. The -, *, and / operations are not permitted on string values. Such operations cause type violations. The function document.write(…..) is allowed to have parameters that are expressions containing string concatenation (+) operations. Using an object name as a parameter is a type rule violation.

**1.4 Treatment of the “
” String**

For simplicity, we will *assume that no string contains
 as a substring, except for the string* ***"
"*** *by its own.* When the interpreter sees “
” (in its exact form) as a parameter in document.write statement, or a variable that has the string value “
”, the semantic action must print a newline, i.e. “\n” in the C statement. All other strings must be printed in the exact way as quoted. Any other similar strings, e.g. “< b r/>“ will be printed as given, just like any other strings.

Since we assume no string constants will contain “
” as a proper substring, we do not allow “
” or any variable that has the string value “
” to be concatenated with another string. **Such an occurrence must be reported as a type rule violation.** A hint would be dealing with “
” differently than normal strings.

1.4 Suggestions on Implementation

To implement type checking and to record the correct variable values, a symbol table is normally implemented. The simple table may need to be organized as two levels to handle objects and their fields. Furthermore, different scopes must be distinguished, usually by following one of the two approaches to be discussed in class.

2. Reporting Type Rule Violations

For simplicity of implementation, the interpreter terminates itself after reporting the first type rule violation it finds in the input program. We will have three sets of test cases. The first set checks on the correctness of the implementation of the grammar rules. The second set checks on the correctness of type checking, with no grammar rule violations in the input. For simplicity, each test case will have no more than one type violations. The third set contains programs that have no violations of grammar rules and type rules. It checks on the correctness of the printed results.

For the first set of test cases, if a syntax error is found, your interpreter does nothing, just like in Project 1. For the second set of test cases, to report the type rule violation, the interpreter must print a single line of error message to the standard output:

Line *<so-and-so>,* type violation

In the above, *<so-and-so>* is the line count in the source code where the type violation is found. For simplicity, no detail of the type violation needs to be reported. After printing the type violation message, **the interpreter terminates itself.**

3. Grading

If your parser for **Part 1** did not pass certain test cases, you must fix your parser to pass such test cases (which are posted on Piazza). These test cases will be slightly modified, e.g. use different variable names, to re-grade your parser.

Overall, points are assigned as following for this project.

- Parsing correctly: 20%
- Correct handling of type rules: 40%
- Correct output by document.write: 40%
- If the Yacc/Bison program has parsing conflicts, deduct 5 pts out of the total.
- Not following submission instructions, deduct 10 pts out of the total.

4. Other Requirements

As in Project 1, the lex/flex program and the yacc/bison program must produce no error messages and warnings from lex/flex/yacc/bison tools. Do not use any precedence defining instructions such as %left to resolve parsing conflicts.

Submission instruction

1) No offline submission (such as email) is accepted.
2) Use the following command at CS lab machines, e.g. the XINU machines (i.e., xinu01.cs ~ xinu20.cs) to submit your homework.
turnin –c cs352 –p p2 [your working directory]

(Your home directories are shared amongst all CS lab machines.)

3) You MUST provide 'Makefile' for every programming question.
a. For example, your 'Makefile' of this assignment may look like

```
parser:y.tab.c lex.yy.c
  gcc y.tab.c lex.yy.c -o parser -lfl
y.tab.c : parser.y
  bison -y -d -g --verbose parser.y
lex.yy.c:parser.l
  lex parser.l
```

```
clean:
    rm -f lex.yy.c y.tab.c
```

NOTE that in the Makefile, a “tab” is required as the first character on each command line, e.g. “<tab> lex paser.l”. Otherwise you will see an error saying “missing separator”.

4) Your program MUST compile and run without any error at CS lab’s Linux machines. Please make sure your Makefile and program runs properly on such machines.
5) Make the final executable program’s name ‘parser’, as shown in the sample Makefile listed above. TA will run your program by executing:
> ./parser program_name > output
Where “program_name” is the file name containing the input program. All printing statements in your semantic actions must print to the standard output.

NOTE: Deviation from the above requirement will get a 10 point of penalty. (For example, if your code receives 90 points, the final score on the Blackboard for this assignment will be 80 points.)

You may find information on the following links posted on piazza to be useful, including the following:
http://ds9a.nl/lex-yacc/cvs/lex-yacc-howto.html

5. Grace Days and Late Turn-In

Each student is granted four grace days (used for projects only) through the rest of the semester. Each grace day allows you to turn in a project no more than 24 hours late. To use a grace day you must inform the TA prior to the project deadline, so that we are sure of your intent. The number of used grace days will be posted on Blackboard for each student.

No other late turn-in requests will be permitted except for highly unusual circumstances such as severe illness as certified by Purdue’s medical facilities. The grace days are intended exactly for handling all short term emergencies, including interview trips and minor illness.